VISIT…

Essential Linux Device Drivers
by Sreekrishnan Venkateswaran

Publisher: Prentice Hall
Pub Date: March 27, 2008
Print ISBN-10: 0-13-239655-6
Print ISBN-13: 978-0-13-239655-4
Pages: 744

Table of Contents
Index

Overview

"Probably the most wide ranging and complete Linux device driver book I've read."

--Alan Cox, Linux Guru and Key Kernel Developer

"Very comprehensive and detailed, covering almost every single Linux device driver type."

--Theodore Ts'o, First Linux Kernel Developer in North America and Chief Platform Strategist of the Linux Foundation

The Most Practical Guide to Writing Linux Device Drivers

Linux now offers an exceptionally robust environment for driver development: with today's kernels, what once required years of development time can be accomplished in days. In this practical, example-driven book, one of the world's most experienced Linux driver developers systematically demonstrates how to develop reliable Linux drivers for virtually any device. *Essential Linux Device Drivers* is for any programmer with a working knowledge of operating systems and C, including programmers who have never written drivers before. Sreekrishnan Venkateswaran focuses on the essentials, bringing together all the concepts and techniques you need, while avoiding topics that only matter in highly specialized situations. Venkateswaran begins by reviewing the Linux 2.6 kernel capabilities that are most relevant to driver developers. He introduces simple device classes; then turns to serial buses such as I2C and SPI; external buses such as PCMCIA, PCI, and USB; video, audio, block, network, and wireless device drivers; user-space drivers; and drivers for embedded Linux–one of today's fastest growing areas of Linux development. For each, Venkateswaran explains the technology, inspects relevant kernel source files, and walks through developing a complete example.

• Addresses drivers discussed in no other book, including drivers for I2C, video, sound, PCMCIA, and different types of flash memory

• Demystifies essential kernel services and facilities, including kernel threads and helper interfaces

• Teaches polling, asynchronous notification, and I/O control

• Introduces the Inter-Integrated Circuit Protocol for embedded Linux drivers

• Covers multimedia device drivers using the Linux-Video subsystem and Linux-Audio framework

• Shows how Linux implements support for wireless technologies such as Bluetooth, Infrared, WiFi, and cellular networking

• Describes the entire driver development lifecycle, through debugging and maintenance

• Includes reference appendixes covering Linux assembly, BIOS calls, and Seq files

Essential Linux Device Drivers
by Sreekrishnan Venkateswaran

Publisher: Prentice Hall
Pub Date: March 27, 2008
Print ISBN-10: 0-13-239655-6
Print ISBN-13: 978-0-13-239655-4
Pages: 744

Table of Contents
Index
Copyright
Prentice Hall Open Source Software Development Series
Foreword
Preface
Acknowledgments
About the Author
Chapter 1. Introduction
- Evolution
- The GNU Copyleft
- Kernel.org
- Mailing Lists and Forums
- Linux Distributions
- Looking at the Sources
- Building the Kernel
- Loadable Modules
- Before Starting

Chapter 2. A Peek Inside the Kernel
- Booting Up
- Kernel Mode and User Mode
- Process Context and Interrupt Context
- Kernel Timers
- Concurrency in the Kernel
- Process Filesystem
- Allocating Memory
- Looking at the Sources

Chapter 3. Kernel Facilities
- Kernel Threads
- Helper Interfaces
- Looking at the Sources

Chapter 4. Laying the Groundwork
- Introducing Devices and Drivers
- Interrupt Handling
- The Linux Device Model
- Memory Barriers
- Power Management
- Looking at the Sources

Chapter 5. Character Drivers
- Char Driver Basics
- Device Example: System CMOS
- Sensing Data Availability
- Talking to the Parallel Port
- RTC Subsystem
- Pseudo Char Drivers
- Misc Drivers
- Character Caveats
- Looking at the Sources

Chapter 6. Serial Drivers
- Layered Architecture
- UART Drivers
- TTY Drivers
- Line Disciplines

Looking at the Sources
Chapter 7. Input Drivers
Input Event Drivers
Input Device Drivers
Debugging
Looking at the Sources
Chapter 8. The Inter-Integrated Circuit Protocol
What's I2C/SMBus?
I2C Core
Bus Transactions
Device Example: EEPROM
Device Example: Real Time Clock
I2C-dev
Hardware Monitoring Using LM-Sensors
The Serial Peripheral Interface Bus
The 1-Wire Bus
Debugging
Looking at the Sources
Chapter 9. PCMCIA and Compact Flash
What's PCMCIA/CF?
Linux-PCMCIA Subsystem
Host Controller Drivers
PCMCIA Core
Driver Services
Client Drivers
Tying the Pieces Together
PCMCIA Storage
Serial PCMCIA
Debugging
Looking at the Sources
Chapter 10. Peripheral Component Interconnect
The PCI Family
Addressing and Identification
Accessing PCI Regions
Direct Memory Access
Device Example: Ethernet-Modem Card
Debugging
Looking at the Sources
Chapter 11. Universal Serial Bus
USB Architecture
Linux-USB Subsystem
Driver Data Structures
Enumeration
Device Example: Telemetry Card
Class Drivers
Gadget Drivers
Debugging
Looking at the Sources
Chapter 12. Video Drivers
Display Architecture
Linux-Video Subsystem
Display Parameters
The Frame Buffer API
Frame Buffer Drivers
Console Drivers
Debugging
Looking at the Sources
Chapter 13. Audio Drivers
Audio Architecture
Linux-Sound Subsystem
Device Example: MP3 Player
Debugging
Looking at the Sources
Chapter 14. Block Drivers

- Storage Technologies
- Linux Block I/O Layer
- I/O Schedulers
- Block Driver Data Structures and Methods
- Device Example: Simple Storage Controller
- Advanced Topics
- Debugging
- Looking at the Sources

Chapter 15. Network Interface Cards
- Driver Data Structures
- Talking with Protocol Layers
- Buffer Management and Concurrency Control
- Device Example: Ethernet NIC
- ISA Network Drivers
- Asynchronous Transfer Mode
- Network Throughput
- Looking at the Sources

Chapter 16. Linux Without Wires
- Bluetooth
- Infrared
- WiFi
- Cellular Networking
- Current Trends

Chapter 17. Memory Technology Devices
- What's Flash Memory?
- Linux-MTD Subsystem
- Map Drivers
- NOR Chip Drivers
- NAND Chip Drivers
- User Modules
- MTD-Utils
- Configuring MTD
- eXecute In Place
- The Firmware Hub
- Debugging
- Looking at the Sources

Chapter 18. Embedding Linux
- Challenges
- Component Selection
- Tool Chains
- Embedded Bootloaders
- Memory Layout
- Kernel Porting
- Embedded Drivers
- The Root Filesystem
- Test Infrastructure
- Debugging

Chapter 19. Drivers in User Space
- Process Scheduling and Response Times
- Accessing I/O Regions
- Accessing Memory Regions
- User Mode SCSI
- User Mode USB
- User Mode I2C
- UIO
- Looking at the Sources

Chapter 20. More Devices and Drivers
- ECC Reporting
- Frequency Scaling
- Embedded Controllers
- ACPI
- ISA and MCA
- FireWire
- Intelligent Input/Output

Amateur Radio
Voice over IP
High-Speed Interconnects
Chapter 21. Debugging Device Drivers
Kernel Debuggers
Kernel Probes
Kexec and Kdump
Profiling
Tracing
Linux Test Project
User Mode Linux
Diagnostic Tools
Kernel Hacking Config Options
Test Equipment
Chapter 22. Maintenance and Delivery
Coding Style
Change Markers
Version Control
Consistent Checksums
Build Scripts
Portable Code
Chapter 23. Shutting Down
Checklist
What Next?
Appendix A. Linux Assembly
Debugging
Appendix B. Linux and the BIOS
Real Mode Calls
Protected Mode Calls
BIOS and Legacy Drivers
Appendix C. Seq Files
The Seq File Advantage
Updating the NVRAM Driver
Looking at the Sources
Index

Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: www.informit.com/ph

Library of Congress Cataloging-in-Publication Data:

Venkateswaran, Sreekrishnan, 1972-
Essential Linux device drivers / Sreekrishnan Venkateswaran.-- 1st ed.
p. cm.
ISBN 0-13-239655-6 (hardback : alk. paper) 1. Linux device drivers (Computer programs)
I. Title.
QA76.76.D49V35 2008
005.4'32--dc22

2008000249

Copyright © 2008 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, v1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

ISBN-13: 978-0-132-39655-4

Text printed in the United States on recycled paper at RR Donnelly in Crawfordsville, IN.

First printing March 2008

Editor-in-Chief
Mark Taub

Executive Editor
Debra Williams Cauley

Managing Editor
Gina Kanouse

Project Edito
Anne Goebel

Copy Editor
Keith Cline

Indexer
Erika Millen

Proofreader
San Dee Phillips

Technical Editors
Vamsi Krishna
Jim Lieb

Publishing Coordinator
Heather Fox

Interior Designer
Laura Robbins

Cover Designer
Alan Clements

Compositor
Molly Sharp

Dedication

This book is dedicated to the ten million visually challenged citizens of India. All author proceeds will go to their cause.

Prentice Hall Open Source Software Development Series

Arnold Robbins, Series Editor

"Real world code from real world applications"

Open Source technology has revolutionized the computing world. Many large-scale projects are in production use worldwide, such as Apache, MySQL, and Postgres, with programmers writing applications in a variety of languages including Perl, Python, and PHP. These technologies are in use on many different systems, ranging from proprietary systems, to Linux systems, to traditional UNIX systems, to mainframes.

The Prentice Hall Open Source Software Development Series is designed to bring you the best of these Open Source technologies. Not only will you learn how to use them for your projects, but you will learn *from* them. By seeing real code from real applications, you will learn the best practices of Open Source developers the world over.

Titles currently in the series include:

Linux® Debugging and Performance Tuning
Steve Best
0131492470, Paper, ©2006

C++ GUI Programming with Qt 4
Jasmin Blanchette, Mark Summerfield
0132354160, Hard, ©2008

The Definitive Guide to the Xen Hypervisor
David Chisnall
013234971X, Hard, ©2008

Understanding AJAX
Joshua Eichorn
0132216353, Paper, ©2007

The Linux Programmer's Toolbox
John Fusco
0132198576, Paper, ©2007

Embedded Linux Primer
Christopher Hallinan
0131679848, Paper, ©2007

The Apache Modules Book
Nick Kew
0132409674, Paper, © 2007

SELinux by Example
Frank Mayer, David Caplan, Karl MacMillan
0131963694, Paper, ©2007

UNIX to Linux® Porting
Alfredo Mendoza, Chakarat Skawratananond,
Artis Walker
0131871099, Paper, ©2006

Rapid Web Applications with TurboGears
Mark Ramm, Kevin Dangoor, Gigi Sayfan
0132433885, Paper, © 2007

Linux Programming by Example
Arnold Robbins
0131429647, Paper, ©2004

The Linux® Kernel Primer
Claudia Salzberg, Gordon Fischer,
Steven Smolski
0131181637, Paper, ©2006

Rapid GUI Programming with Python and Qt
Mark Summerfield
0132354187, Hard, © 2008

Essential Linux Device Drivers
Sreekrishnan Venkateswaran
0132396556, Hard, ©2008

New to the series: Digital Short Cuts

Short Cuts are short, concise, PDF documents designed specifically for busy technical professionals like you. Each Short Cut is tightly focused on a specific technology or technical problem. Written by industry experts and best selling authors, Short Cuts are published with you in mind — getting you the technical information that you need — now.

Understanding AJAX:
Consuming the Sent Data with XML and JSON
Joshua Eichorn
0132337932, Adobe Acrobat PDF, © 2007

Debugging Embedded Linux
Christopher Hallinan
0131580132, Adobe Acrobat PDF, © 2007

Using BusyBox
Christopher Hallinan
0132335921, Adobe Acrobat PDF, © 2007

Foreword

If you're holding this book, you may be asking yourself: Why "yet another" Linux device driver book? Aren't there already a bunch of them?

The answer is: This book is a quantum leap ahead of the others.

First, it is up-to-date, covering recent 2.6 kernels. Second, and more important, this book is *thorough*. Most device driver books just cover the topics described in standard Unix internals books or operating system books, such as serial lines, disk drives, and filesystems, and, if you're lucky, the networking stack.

This book goes much further; it doesn't shy away from the hard stuff that you have to deal with on modern PC and embedded hardware, such as PCMCIA, USB, I^2C, video, audio, flash memory, wireless communications, and so on. You name it, if the Linux kernel talks to it, then this book tells you about it.

No stone is left unturned; no dark corner is left unilluminated.

Furthermore, the author has earned his stripes: It's a thrill ride just to read his description of putting Linux on a wristwatch in the late 1990s!

I'm pleased and excited to have this book as part of the Prentice Hall Open Source Software Development Series. It is a shining example of the exciting things happening in the Open Source world. I hope that you will find here what you need for your work on the kernel, and that you will enjoy the process, too!

Arnold Robbins
Series Editor

Preface

It was the late 1990s, and at IBM we were putting the Linux kernel on a wristwatch. The target device was tiny, but the task was turning out to be tough. The Memory Technology Devices subsystem didn't exist in the kernel, which meant that before a filesystem could start life on the watch's flash memory, we had to develop the necessary storage driver from scratch. Interfacing the watch's touch screen with user applications was complicated because the kernel's *input* event driver interface hadn't been conceived yet. Getting X Windows to run on the watch's LCD wasn't easy because it didn't work well with frame buffer drivers. Of what use is a waterproof Linux wristwatch if you can't stream stock quotes from your bathtub? Bluetooth integration with Linux was several years away, and months were spent porting a proprietary Bluetooth stack to Internet-enable the watch. Power management support was good enough only to squeeze a few hours of juice from the watch's battery; hence we had work cut out on that front, too. Linux-Infrared was still unstable, so we had to coax the stack before we could use an Infrared keyboard for data entry. And we had to compile the compiler and cross-compile a compact application-set because there were no accepted distributions in the consumer electronics space.

Fast forward to the present: The baby penguin has grown into a healthy teenager. What took thousands of lines of code and a year in development back then can be accomplished in a few days with the current kernels. But to become a versatile kernel engineer who can magically weave solutions, you need to understand the myriad features and facilities that Linux offers today.

About the Book

Among the various subsystems residing in the kernel source tree, the *drivers/* directory constitutes the single largest chunk and is several times bigger than the others. With new and diverse technologies arriving in popular form factors, the development of new device drivers in the kernel is accelerating steadily. The latest kernels support more than 70 device driver families.

This book is about writing Linux device drivers. It covers the design and development of major device classes supported by the kernel, including those I missed during my Linux-on-Watch days. The discussion of each driver family starts by looking at the corresponding technology, moves on to develop a practical example, and ends by looking at relevant kernel source files. Before foraying into the world of device drivers, however, this book introduces you to the kernel and discusses the important features of 2.6 Linux, emphasizing those portions that are of special interest to device driver writers.

Audience

This book is intended for the intermediate-level programmer eager to tweak the kernel to enable new devices. You should have a working knowledge of operating system concepts. For example, you should know what a system call is and why concurrency issues have to be factored in while writing kernel code. The book assumes that you have downloaded Linux on your system, poked through the kernel sources, and at least skimmed through some related documentation. And you should be pretty good in C.

Summary of Chapters

The first 4 chapters prepare you to digest the rest of the book. The next 16 chapters discuss drivers for different device families. A chapter that describes device driver debugging techniques comes next. The penultimate chapter provides perspective on maintenance and delivery. We shut down by walking through a checklist that summarizes how to set forth on your way to Linux-enablement when you get hold of a new device.

Chapter 1, "Introduction," starts our tryst with Linux. It hurries you through downloading the kernel sources, making trivial code changes, and building a bootable kernel image.

Chapter 2, "A Peek Inside the Kernel," takes a brisk look into the innards of the Linux kernel and teaches you some must-know kernel concepts. It first takes you through the boot process and then describes kernel services particularly relevant to driver development, such as kernel timers, concurrency management, and memory allocation.

Chapter 3, "Kernel Facilities," examines several kernel services that are useful components in the toolbox of driver developers. The chapter starts by looking at kernel threads, which is a way to implement background tasks inside the kernel. It then moves on to helper interfaces such as linked lists, work queues, completion functions, and notifier chains. These helper facilities simplify your code, weed out redundancies from the kernel, and help long-term maintenance.

Chapter 4, "Laying the Groundwork," builds the foundation for mastering the art of writing Linux device drivers. It introduces devices and drivers by giving you a bird's-eye view of the architecture of a typical PC-compatible system and an embedded device. It then looks at basic driver concepts such as interrupt handling and the kernel's device model.

Chapter 5, "Character Drivers," looks at the architecture of character device drivers. Several concepts introduced in this chapter, such as polling, asynchronous notification, and I/O control, are relevant to subsequent chapters, too, because many device classes discussed in the rest of the book are "super" character devices.

Chapter 6, "Serial Drivers," explains the kernel layer that handles serial devices.

Chapter 7, "Input Drivers," discusses the kernel's input subsystem that is responsible for servicing devices such as keyboards, mice, and touch-screen controllers.

Chapter 8, "The Inter-Integrated Circuit Protocol," dissects drivers for devices such as EEPROMs that are connected to a system's I^2C bus or SMBus. This chapter also looks at other serial interfaces such as SPI bus and 1-wire bus.

Chapter 9, "PCMCIA and Compact Flash," delves into the PCMCIA subsystem. It teaches you to write drivers for devices having a PCMCIA or Compact Flash form factor.

Chapter 10, "Peripheral Component Interconnect," looks at kernel support for PCI and its derivatives.

Chapter 11, "Universal Serial Bus," explores USB architecture and explains how you can use the services of the Linux-USB subsystem to write drivers for USB devices.

Chapter 12, "Video Drivers," examines the Linux-Video subsystem. It finds out the advantages offered by the frame buffer abstraction and teaches you to write frame buffer drivers.

Chapter 13, "Audio Drivers," describes the Linux-Audio framework and explains how to implement audio drivers.

Chapter 14, "Block Drivers," focuses on drivers for storage devices such as hard disks. In this chapter, you also learn about the different I/O schedulers supported by the Linux-Block subsystem.

Chapter 15, "Network Interface Cards," is devoted to network device drivers. You learn about kernel networking data structures and how to interface network drivers with protocol layers.

Chapter 16, "Linux Without Wires," looks at driving different wireless technologies such as Bluetooth, Infrared, WiFi, and cellular communication.

Chapter 17, "Memory Technology Devices," discusses flash memory enablement on embedded devices. The chapter ends by examining drivers for the Firmware Hub found on PC systems.

Chapter 18, "Embedding Linux," steps into the world of embedded Linux. It takes you through the main firmware components of an embedded solution such as bootloader, kernel, and device drivers. Given the soaring popularity of Linux in the embedded space, it's more likely that you will use the device driver skills that you

acquire from this book to enable embedded systems.

Chapter 19, "Drivers in User Space," looks at driving different types of devices from user space. Some device drivers, especially ones that are heavy on policy and light on performance requirements, are better off residing in user land. This chapter also explains how the Linux process scheduler affects the response times of user mode drivers.

Chapter 20, "More Devices and Drivers," takes a tour of a potpourri of driver families not covered thus far, such as *Error Detection And Correction* (EDAC), FireWire, and ACPI.

Chapter 21, "Debugging Device Drivers," teaches about different types of debuggers that you can use to debug kernel code. In this chapter, you also learn to use trace tools, kernel probes, crash-dump, and profilers. When you develop a driver, be armed with the driver debugging skills that you learn in this chapter.

Chapter 22, "Maintenance and Delivery," provides perspective on the software development life cycle.

Chapter 23, "Shutting Down," takes you through a checklist of work items when you embark on Linux-enabling a new device. The book ends by pondering *What next?*

Device drivers sometimes need to implement code snippets in assembly, so Appendix A, "Linux Assembly," takes a look at the different facets of assembly programming on Linux. Some device drivers on x86-based systems depend directly or indirectly on the BIOS, so Appendix B, "Linux and the BIOS," teaches you how Linux interacts with the BIOS. Appendix C, "Seq Files," describes seq files, a kernel helper interface introduced in the 2.6 kernel that device drivers can use to monitor and trend data points.

The book is generally organized according to device and bus complexity, coupled with practical reasons of dependencies between chapters. So, we start off with basic device classes such as character, serial, and input. Next, we look at simple serial buses such as I^2C and SMBus. External I/O buses such as PCMCIA, PCI, and USB follow. Video, audio, block, and network devices usually interface with the processor via these I/O buses, so we look at them soon after. The next portions of the book are oriented toward embedded Linux and cover technologies such as wireless networking and flash memory. User-space drivers are discussed toward the end of the book.

Kernel Version

This book is generally up to date as of the 2.6.23/2.6.24 kernel versions. Most code listings in this book have been tested on a 2.6.23 kernel. If you are using a later version, look at Linux websites such as lwn.net to learn about the kernel changes since 2.6.23/24.

Book Website

I've set up a website at elinuxdd.com to provide updates, errata, and other information related to this book.

Conventions Used

Source code, function names, and shell commands are written `like this`. The shell prompt used is **bash>**. Filename are written in italics, *like this.* Italics are also used to introduce new terms.

Some chapters modify original kernel source files while implementing code examples. To clearly point out the changes, newly inserted code lines are prefixed with **+**, and any deleted code lines with `-`.

Sometimes, for simplicity, the book uses generic references. So if the text points you to the *arch/your-arch/* directory, it should be translated, for example, to *arch/x86/* if you are compiling the kernel for the x86 architecture. Similarly, any mention of the *include/asm-your-arch/* directory should be read as *include/asm-arm/* if you are, for instance, building the kernel for the ARM architecture. The * symbol and *X* are occasionally used as wildcard characters in filenames. So, if a chapter asks you to look at *include/linux/time*.h*, look at the

header files, *time.h, timer.h, times.h*, and *timex.h* residing in the *include/linux/* directory. If a section talks about */dev/input/eventX* or */sys/devices/platform/i8042/serioX/, X* is the interface number that the kernel assigns to your device in the context of your system configuration.

The ⟶ symbol is sometimes inserted between command or kernel output to attach explanations.

Simple regular expressions are occasionally used to compactly list function prototypes. For example, the section "Direct Memory Access" in Chapter 10, "Peripheral Component Interconnect," refers to `pci_[map|unmap|dma_sync]_single()` instead of explicitly citing `pci_map_single()`, `pci_umap_single()`, and `pci_dma_sync_single()`.

Several chapters refer you to user-space configuration files. For example, the section that describes the boot process opens */etc/rc.sysinit*, and the chapter that discusses Bluetooth refers to */etc/bluetooth/pin*. The exact names and locations of such files might, however, vary according to the Linux distribution you use.

Acknowledgments

First, I raise my hat to my editors at Prentice Hall: Debra Williams Cauley, Anne Goebel, and Keith Cline. Without their supporting work, this book would not have materialized. I thank Mark Taub for his interest in this project and for initiating it.

Several sources have contributed to my learning in the past decade: the many teammates with whom I worked on Linux projects, the mighty kernel sources, mailing lists, and the Internet. All these have played a part in helping me write this book.

Martin Streicher of *Linux Magazine* changed me from a full-time coder to a spare-time writer when he offered me the magazine's "Gearheads" kernel column. I gratefully acknowledge the many lessons in technical writing that I've learned from him.

I owe a special debt of gratitude to my technical reviewers. Vamsi Krishna patiently read through each chapter of the manuscript. His numerous suggestions have made this a better book. Jim Lieb provided valuable feedback on several chapters. Arnold Robbins reviewed the first few chapters and provided insightful comments.

Finally, I thank my parents and my wife for their love and support. And thanks to my baby daughter for constantly reminding me to spend cycles on the book by her wobbly walk that bears an uncanny resemblance to that of a penguin.

About the Author

Sreekrishnan Venkateswaran has a master's degree in computer science from the Indian Institute of Technology, Kanpur, India. During the past 12 years that he has been working for IBM, he has ported Linux to various embedded devices such as a wristwatch, handheld, music player, VoIP phone, pacemaker programmer, and remote patient monitoring system. Sreekrishnan was a contributing editor and kernel columnist to the *Linux Magazine* for more than 2 years. Currently, he manages the embedded solutions group at IBM India.

Chapter 1. Introduction

In This Chapter

- Evolution 2
- The GNU Copyleft 3
- Kernel.org 4
- Mailing Lists and Forums 4
- Linux Distributions 5
- Looking at the Sources 6
- Building the Kernel 10
- Loadable Modules 12
- Before Starting 14

Linux lures. It has the enticing aroma of an internationalist project where people of all nationalities, creed, and gender collaborate. Free availability of source code and a well-understood UNIX-like application programming environment have contributed to its runaway success. High-quality support from experts available instantly over the Internet at no charge has also played a major role in stitching together a huge Linux community.

Developers get incredibly excited about working on technologies where they have access to all the sources because that lets them create innovative solutions. You can, for example, hack the sources and customize Linux to boot in a few seconds on your device, a feat that is hard to achieve with a proprietary operating system.

Evolution

Linux started as the hobby of a Finnish college student named Linus Torvalds in 1991, but quickly metamorphed into an advanced operating system popular all over the planet. From its first release for the Intel 386 processor, the kernel has gradually grown in complexity to support numerous architectures, multiprocessor hardware, and high-performance clusters. The full list of supported CPUs is long, but some of the major supported architectures are x86, IA64, ARM, PowerPC, Alpha, s390, MIPS, and SPARC. Linux has been ported to hundreds of hardware platforms built around these processors. The kernel is continuously getting better, and the evolution is progressing at a frantic pace.

Although it started life as a desktop-operating system, Linux has penetrated the embedded and enterprise worlds and is touching our daily lives. When you push the buttons on your handheld, flip your remote to the weather channel, or visit the hospital for a physical checkup, it's increasingly likely that some Linux code is being set into motion to come to your service. Linux's free availability is helping its evolution as much as its technical superiority. Whether it's an initiative to develop sub-$100 computers to enable the world's poor or pricing pressure in the consumer electronics space, Linux is today's operating system of choice, because proprietary operating systems sometimes cost more than the desired price of the computers themselves.

Chapter 1. Introduction

In This Chapter

- Evolution 2
- The GNU Copyleft 3
- Kernel.org 4
- Mailing Lists and Forums 4
- Linux Distributions 5
- Looking at the Sources 6
- Building the Kernel 10
- Loadable Modules 12
- Before Starting 14

Linux lures. It has the enticing aroma of an internationalist project where people of all nationalities, creed, and gender collaborate. Free availability of source code and a well-understood UNIX-like application programming environment have contributed to its runaway success. High-quality support from experts available instantly over the Internet at no charge has also played a major role in stitching together a huge Linux community.

Developers get incredibly excited about working on technologies where they have access to all the sources because that lets them create innovative solutions. You can, for example, hack the sources and customize Linux to boot in a few seconds on your device, a feat that is hard to achieve with a proprietary operating system.

Evolution

Linux started as the hobby of a Finnish college student named Linus Torvalds in 1991, but quickly metamorphed into an advanced operating system popular all over the planet. From its first release for the Intel 386 processor, the kernel has gradually grown in complexity to support numerous architectures, multiprocessor hardware, and high-performance clusters. The full list of supported CPUs is long, but some of the major supported architectures are x86, IA64, ARM, PowerPC, Alpha, s390, MIPS, and SPARC. Linux has been ported to hundreds of hardware platforms built around these processors. The kernel is continuously getting better, and the evolution is progressing at a frantic pace.

Although it started life as a desktop-operating system, Linux has penetrated the embedded and enterprise worlds and is touching our daily lives. When you push the buttons on your handheld, flip your remote to the weather channel, or visit the hospital for a physical checkup, it's increasingly likely that some Linux code is being set into motion to come to your service. Linux's free availability is helping its evolution as much as its technical superiority. Whether it's an initiative to develop sub-$100 computers to enable the world's poor or pricing pressure in the consumer electronics space, Linux is today's operating system of choice, because proprietary operating systems sometimes cost more than the desired price of the computers themselves.

The GNU Copyleft

The GNU project (GNU is a recursive acronym for *GNU's Not UNIX*) predates Linux and was launched to develop a free UNIX-like operating system. A complete GNU operating system is powered by the Linux kernel but also contains components such as libraries, compilers, and utilities. A Linux-based computer is hence more accurately a GNU/Linux system. All components of a GNU/Linux system are built using free software.

There are different flavors of free software. One such flavor is called *public domain* software. Software released under the public domain is not copyrighted, and no restrictions are imposed on its usage. You can use it for free, make changes to it, and even restrict the distribution of your modified sources. As you can see, the "no restrictions" clause introduces the power to introduce restrictions downstream.

The Free Software Foundation, the primary sponsor of the GNU project, created the *GNU Public License* (GPL), also called a *copyleft*, to prevent the possibility of middlemen transforming free software into proprietary software. Those who modify copylefted software are required to also copyleft their derived work. The Linux kernel and most components of a GNU system such as the *GNU Compiler Collection* (GCC) are released under the GPL. So, if you make modifications to the kernel, you have to return your changes back to the community. Essentially, you have to pass on the rights vested on you by the copyleft.

> The Linux kernel is licensed under GPL version 2. There is an ongoing debate in the kernel community about whether the kernel should move to GPLv3, the latest version of the GPL. The current tide seems to be against relicensing the kernel to adopt GPLv3.

Linux applications that invoke system calls to access kernel services are not considered derived work, however, and won't be restricted by the GPL. Similarly, libraries are covered by a less-stringent license called the *GNU Lesser General Public License* (LGPL). Proprietary software is permitted to dynamically link with libraries released under the LGPL.

Kernel.org

The primary repository of Linux kernel sources is www.kernel.org. The website contains all released kernel versions. A number of websites around the world mirror the contents of kernel.org.

In addition to released kernels, kernel.org also hosts a set of *patches* maintained by front-line developers that serve as a test bed for future stable releases. A patch is a text file containing source code differences between a development tree and the original snapshot from which the developer started work. A popular patch-set available at kernel.org is the `-mm` patch periodically released by Andrew Morton, the lead maintainer of the Linux kernel. You will find experimental features in the `-mm` patch that have not yet made it to the mainline source tree. Another patch-set periodically released on kernel.org is the `–rt` (real time) patch maintained by Ingo Molnar. Several `–rt` features have been merged into the mainline kernel.

Mailing Lists and Forums

The *Linux Kernel Mailing List* (LKML) is the forum where developers debate on design issues and decide on future features. You can find a real-time feed of the mailing list at www.lkml.org. The kernel now contains several million lines of code contributed by thousands of developers all over the world. LKML acts as the thread that ties all these developers together.

LKML is not for general Linux questions. The basic rule is to post only questions pertaining to kernel development that have not been previously answered in the mailing list or in popularly available documentation. If the C compiler crashed while compiling your Linux application, you should post that question elsewhere.

Discussions in some LKML threads are more interesting than a *New York Times* bestseller. Spend a few hours browsing LKML archives to get an insight into the philosophy behind the Linux kernel.

Most subprojects in the kernel have their own specific mailing lists. So, subscribe to the linux-mtd mailing list if you are developing a Linux flash memory driver or initiate a thread in the linux-usb-devel mailing list if you think you have found a bug in the USB mass storage driver. We refer to relevant mailing lists at the end of several chapters.

In various forums, kernel experts from around the globe gather under one roof. The *Linux Symposium* held annually at Ottawa, Canada, is one such conference. Others include the *Linux Kongress* that takes place in Germany and *linux.conf.au* organized in Australia. There are also numerous commercial Linux forums where industry leaders meet and share their insights. An example is the *LinuxWorld Conference and Expo* held yearly in North America.

For the latest news from the developer community, check out http://lwn.net/. If you want to glean the highlights of the latest kernel release without many cryptic references to kernel internals, this might be a good place to look. You can find another web community that discusses current kernel topics at http://kerneltrap.org/.

With every major kernel release, you will see sweeping improvements, be it kernel preemption, lock-free readers, new services to offload work from interrupt handlers, or support for new architectures. Stay in constant touch with the mailing lists, websites, and forums, to keep yourself in the thick of things.

Linux Distributions

Because a GNU/Linux system consists of numerous utilities, programs, libraries, and tools, in addition to the kernel, it's a daunting task to acquire and correctly install all the pieces. Linux distributions come to the rescue by classifying the components and bundling them into packages in an orderly fashion. A typical distribution contains thousands of ready-made packages. You need not worry about downloading the right program versions or fixing dependency issues.

Because packaging is a way to make a lot of money within the ambit of the GNU license, there are several Linux distributions in the market today. Distributions such as Red Hat/Fedora, Debian, SuSE, Slackware, Gentoo, Ubuntu, and Mandriva are primarily meant for the desktop user. MontaVista, TimeSys, and Wind River distributions are geared toward embedded development. Embedded Linux distributions also include a dynamically configurable compact application-set to tailor the system's footprint to suit resource constraints.

In addition to packaging, distributions offer value-adds for kernel development. Many projects start development based on kernels supplied by a distribution rather than a kernel released officially at kernel.org. Reasons for this include the following:

- Linux distributions that comply with standards relevant to your device's industry domain are often better starting points for development. *Special Interest Groups* (SIGs) have taken shape to promote Linux in various domains. The *Consumer Electronics Linux Forum* (CELF), hosted at www.celinuxforum.org, focuses on using Linux on consumer electronics devices. The CELF specification defines the support level of features such as scalable footprint, fast boot, execute in place, and power management, desirable on consumer electronics devices. The efforts of the *Open Source Development Lab* (OSDL), hosted at www.osdl.org, centers on characteristics distinct to carrier-grade devices. OSDL's *Carrier Grade Linux* (CGL) specification codifies value additions such as reliability, high availability, runtime patching, and enhanced error recovery, important in the telecom space.

- The mainline kernel might not include full support for the embedded controller of your choice even if the controller is built around a kernel-supported CPU core. A Linux distribution might offer device drivers for all the peripheral modules inside the controller, however.

- Debugging tools that you plan to use during kernel development may not be part of the mainline kernel. For example, the kernel has no built-in debugger support. If you want to use a kernel debugger during development, you have to separately download and apply the corresponding patches. You have to endure more hassles if tested patches are not readily available for your kernel version. Distributions prepackage many useful debugging features, so you can start using them right away.

- Some distributions provide legal indemnification so that your company won't be liable for lawsuits arising out of kernel bugs.

- Distributions tend to do a lot of testing on the kernels they release.[1]

  [1] Because this necessitates freezing the kernel to a version that is not the latest, distribution-supplied kernels often contain back-ports of some features released in later official kernels.

- You can purchase service and support packages from distribution vendors for kernels that they supply.

Looking at the Sources

Before we start wetting our toes in the kernel, let's download the sources, learn to apply a patch, and look at the layout of the code tree.

First, go to www.kernel.org and get the latest stable tree. The sources are archived as *tar* files compressed in both *gzip* (.gz) and *bzip2* (.bz2) formats. Obtain the source files by uncompressing and untarring the zipped tar ball. In the following commands, replace X.Y.Z with the latest kernel version, such as 2.6.23:

```
bash> cd /usr/src
bash> wget www.kernel.org/pub/linux/kernel/vX.Y/linux-X.Y.Z.tar.bz2
...
bash> tar xvfj linux-X.Y.Z.tar.bz2
```

Now that you have the unpacked source tree in */usr/src/linux-X.Y.Z/* on your system, let's enable some experimental test features into the tree by getting a corresponding `-mm` (Andrew Morton) patch:

Code View:

```
bash> cd /usr/src
bash> wget www.kernel.org/pub/linux/kernel/people/akpm/patches/X.Y/X.Y.Z/X.Y.Z-
mm2/X.Y.Z-mm2.bz2
```

Apply the patch:

```
bash> cd /usr/src/linux-X.Y.Z/
bash> bzip2 -dc ../X.Y.Z-mm2.bz2 | patch -p1
```

The `-dc` option asks bzip2 to uncompress the specified files to standard output. This is piped to the *patch* utility, which applies changes to each modified file in the code tree.

If you need to apply multiple patches, do so in the right sequence. To generate a kernel enabled with the `X.Y.Z-aa-bb` patch, first download the full `X.Y.Z` kernel sources, apply the `X.Y.Z-aa` patch, and then apply the `X.Y.Z-aa-bb` patch.

Patch Submission

To generate a kernel patch out of your changes, use the `diff` command:

Code View:
```
bash> diff –Nur /path/to/original/kernel /path/to/your/kernel > changes.patch
```

Note that the original kernel precedes the changed version in the `diff`-ing order. As per 2.6 kernel patch submission conventions, you also need to add a line at the end of the patch that says this:

```
Signed-off-by: Name <Email>
```

With this, you certify that you wrote the code yourself and that you have the right to contribute it.

You are now all set to post your patch to the relevant mailing list, such as LKML.

Look at *Documentation/SubmittingPatches* for a guide on creating patches for submission and at *Documentation/applying-patches.txt* for a tutorial on applying patches.

Now that your patched */usr/src/linux-X.Y.Z/* tree is ready for use, let's take a moment to observe how the source layout is organized. Go to the root of the source tree and list its contents. The directories branching out from the root of the code tree are as follows:

1. *arch.* This directory contains architecture-specific files. You will see separate subdirectories under *arch/* for processors such as ARM, Motorola 68K, s390, MIPS, Alpha, SPARC, and IA64.

2. *block.* This primarily contains the implementation of I/O scheduling algorithms for block storage devices.

3. *crypto.* This directory implements cipher operations and the cryptographic API, used, for example, by some WiFi device drivers for implementing encryption algorithms.

4. *Documentation.* This directory has brief descriptions of various kernel subsystems. This can be your first stop to dig for answers to kernel-related queries.

5. *drivers.* Device drivers for numerous device classes and peripheral controllers reside in this directory. The device classes include character, serial, *Inter-Integrated Circuit* (I^2C), *Personal Computer Memory Card International Association* (PCMCIA), *Peripheral Component Interconnect* (PCI), *Universal Serial Bus* (USB), video, audio, block, *Integrated Drive Electronics* (IDE), *Small Computer System Interface* (SCSI), CD-ROM, network adapters, *Asynchronous Transfer Mode* (ATM), Bluetooth, and *Memory Technology Devices* (MTD). Each of these classes live in a separate subdirectory under *drivers/*. You will, for instance, find PCMCIA driver sources inside the *drivers/pcmcia/* directory and MTD drivers inside the *drivers/mtd/* directory. The subdirectories present under *drivers/* constitute the primary subjects for this book.

6. *fs.* This directory contains the implementation of filesystems such as EXT3, EXT4, reiserfs, FAT, VFAT, sysfs, procfs, isofs, JFFS2, XFS, NTFS, and NFS.

7. *include.* Kernel header files live here. Subdirectories prefixed with *asm* contain headers specific to the particular architecture. So the directory *include/asm-x86/* contains header files pertaining to the x86 architecture, whereas *include/asm-arm/* holds headers for the ARM architecture.

8. *init.* This directory contains high-level initialization and startup code.

9. *ipc.* This contains support for *Inter-Process Communication* (IPC) mechanisms such as message queues, semaphores, and shared memory.

10. *kernel.* The architecture-independent portions of the base kernel can be found here.

11. *lib.* Library routines such as generic *kernel object* (kobject) handlers and *Cyclic Redundancy Code* (CRC) computation functions stay here.

12. *mm.* The memory management implementation lives here.

13. *net.* Networking protocols reside under this directory. Protocols implemented include *Internet Protocol version 4* (IPv4), IPv6, *Internetwork Protocol eXchange* (IPX), Bluetooth, ATM, Infrared, *Link Access Procedure Balanced* (LAPB), and *Logical Link Control* (LLC).

14. *scripts.* Scripts used during kernel build reside here.

15. *security.* This directory contains the framework for security.

16. *sound.* The Linux audio subsystem is based in this directory.

17. *usr.* This currently contains the *initramfs* implementation.

Unified x86 Architecture Tree

Starting with the 2.6.24 kernel release, the i386 and the x86_64 (the 64-bit cousin of the 32-bit i386) architecture-specific trees have been unified into a common *arch/x86/* directory. If you are using a pre-2.6.24 kernel, replace references to *arch/x86/* in this book with *arch/i386/*. Similarly, change any occurrence of *include/asm-x86/* to *include/asm-i386/*. Some filenames within these directories have also changed.

Wading through these large directories in search of symbols and other code elements can be a tough task. The tools in Table 1.1 are worthy aids as you navigate the kernel source tree.

Table 1.1. Tools That Aid Source Tree Navigation

Tool	Description
lxr	The Linux cross-referencer, lxr, downloadable from http://lxr.sourceforge.net/, lets you traverse the kernel tree using a web browser by providing hyperlinks to connect kernel symbols with their definitions and uses.
cscope	cscope, hosted at http://cscope.sourceforge.net/, builds a symbolic database from all files in a source tree, so you can quickly locate declarations, definitions, regular expressions, and more. Cscope might not be as versatile as lxr, but it gives you the flexibility of using the search features of your favorite text editor rather than a browser. From the root of your kernel tree, issue the `cscope -qkRv` command to build the cross-reference database. The `-q` option generates more indexing information, so searches become noticeably faster at the expense of extra initial startup time. The `–k` option requests cscope to tune its behavior to suit kernel sources, `-R` asks for recursive subdirectory traversal, and `–v` appeals for verbose messages. You can obtain the detailed invocation syntax from the man page.
ctags/etags	The ctags utility, downloadable from http://ctags.sourceforge.net/, generates cross-reference tags for many languages, so you can locate symbol and function definitions in a source tree from within an editor such as vi. Do `make tags` from the root of your kernel tree to ctag all source files. The etags utility generates similar indexing information understood by the emacs editor. Issue `make TAGS` to etag your kernel source files.
Utilities	Tools such as grep, find, sdiff, strace, od, dd, make, tar, file, and objdump.
GCC options	You may ask GCC to generate preprocessed source code using the `-E` option. Preprocessed code contains header file expansions and reduces the need to hop-skip through nested include files to expand multiple levels of macros. Here is a usage example to preprocess *drivers/char/mydrv.c* and produce expanded output in *mydrv.i:* **`bash> gcc -E drivers/char/mydrv.c -D__KERNEL__ -Iinclude -Iinclude/asm-x86/mach-default > mydrv.i`** The include paths specified using the `-I` option depend on the header files included by your code. GCC generates assembly listings if you use the `-S` option. To generate an assembly listing in *mydrv.s* for *drivers/char/mydrv.c*, do this: **`bash> gcc -S drivers/char/mydrv.c -D__KERNEL__ -Iinclude -Ianother/include/path`**

Building the Kernel

Now that you have an idea of the source tree layout, let's make a trivial code change, compile, and get it running. Go to the top-level *init/* directory and venture to make a small code change to the initialization file *main.c*. Add a print statement to the beginning of the function, `start_kernel()`, declaring your love for polar bears:

```
asmlinkage void __init start_kernel(void)
{
    char *command_line;
    extern struct kernel_param __start___param[],
           __stop___param[];

+   printk("Penguins are cute, but so are polar bears\n");

    /* ... */

    rest_init();
}
```

You're now ready to kick off the build process. Go to the root of the source tree and start with a clean slate:

```
bash> cd /usr/src/linux-X.Y.Z/
bash> make clean
```

Configure the kernel. This is when you pick and choose the pieces that form part of the operating system. You may specify whether each desired component is to be statically or dynamically linked to the kernel:

```
bash> make menuconfig
```

`menuconfig` is a text interface to the kernel configuration menu. Use `make xconfig` to get a graphical interface. The configuration information that you choose is saved in a file named *.config* in the root of your source tree. If you don't want to weave the configuration from scratch, use the file *arch/your-arch/defconfig* (or *arch/your-arch/configs/your-machine_defconfig* if there are several supported platforms for your architecture) as the starting point. So, if you are compiling the kernel for the 32-bit x86 architecture, do this:

```
bash> cp arch/x86/configs/i386_defconfig .config
```

Compile the kernel and generate a compressed boot image:

```
bash> make bzImage
```

The kernel image is produced in *arch/x86/boot/bzImage.* Update your boot partition:

```
bash> cp arch/x86/boot/bzImage /boot/vmlinuz
```

You might need to alert your bootloader about the arrival of the new boot image. If you are using the GRUB bootloader, it figures this out automatically; but if you are using LILO, raise a flag:

```
bash> /sbin/lilo
Added linux *
```

Finally, restart the machine and boot in to your new kernel:

```
bash> reboot
```

The first message in the boot sequence launches your campaign for polar bears.

Loadable Modules

Because Linux runs on a variety of architectures and supports zillions of I/O devices, it's not feasible to permanently compile support for all possible devices into the base kernel. Distributions generally package a minimal kernel image and supply the rest of the functionalities in the form of *kernel modules.* During runtime, the necessary modules are dynamically loaded on demand.

To generate modules, go to the root of your kernel source tree and build:

```
bash> cd /usr/src/linux-X.Y.Z/
bash> make modules
```

To install the produced modules, do this:

```
bash> make modules_install
```

This creates a kernel source directory structure under */lib/modules/X.Y.Z/kernel/* and populates it with loadable module objects. This also invokes the depmod utility that generates module dependencies in the file */lib/modules/X.Y.Z/modules.dep*.

The following utilities are available to manipulate modules: *insmod*, *rmmod*, *lsmod, modprobe*, *modinfo*, and *depmod.* The first two are utilities to insert and remove modules, whereas lsmod lists the modules that are currently loaded. modprobe is a cleverer version of insmod that also inserts dependent modules after examining the contents of */lib/modules/X.Y.Z/modules.dep.* For example, assume that you need to mount a *Virtual File Allocation Table* (VFAT) partition present on a USB pen drive. Use modprobe to load the VFAT filesystem driver:[2]

> [2] This example assumes that the module is not autoloaded by the kernel. If you enable *Automatic Kernel Module Loading* (`CONFIG_KMOD`) during configuration, the kernel automatically runs modprobe with the appropriate arguments when it detects missing subsystems. You'll learn about module autoloading in Chapter 4, "Laying the Groundwork."

```
bash> modprobe vfat
bash> lsmod
Module       Size     Used by
vfat         14208    0
fat          49052    1 vfat
nls_base     9728     2 vfat, fat
```

As you see in the lsmod output, modprobe inserts three modules rather than one. modprobe first figures out that it has to insert */lib/modules/X.Y.Z/kernel/fs/vfat/vfat.ko.* But when it peeks into the dependency file */lib/modules/X.Y.Z/modules.dep*, it finds the following line and realizes that it has to load two other dependent modules first:

```
/lib/modules/X.Y.Z/kernel/fs/vfat/vfat.ko:
/lib/modules/X.Y.Z/kernel/fs/fat/fat.ko
/lib/modules/X.Y.Z/kernel/fs/nls/nls_base.ko
```

It then proceeds to load *fat.ko* and *nls_base.ko* before attempting to insert *vfat.ko*, thus automatically loading all the modules you need to mount your VFAT partition.

Use the modinfo utility to extract verbose information about the modules you just loaded:

```
bash> modinfo vfat
filename:       /lib/modules/X.Y.Z/kernel/fs/vfat/vfat.ko
license:        GPL
description:    VFAT filesystem support
...
depends:        fat, nls_base
```

To compile a kernel driver as a module, toggle the corresponding menu choice button to <M> while configuring the kernel. Most of the device driver examples in this book are implemented as kernel modules. To build a module *mymodule.ko* from its source file *mymodule.c*, create a one-line Makefile and execute it as follows:

```
bash> cd /path/to/module-source/
bash> echo "obj-m += mymodule.ko" > Makefile
bash> make –C /path/to/kernel-sources/ M=`pwd` modules
make: Entering directory '/path/to/kernel-sources'
  Building modules, stage 2.
  MODPOST
  CC /path/to/module-sources/mymodule.mod.o
  LD [M] /path/to/module-sources/mymodule.ko
make: Leaving directory '/path/to/kernel-sources'
bash> insmod ./mymodule.ko
```

Kernel modules render the kernel footprint smaller and the develop-build-test cycle shorter. You only need to recompile the particular module and reinsert it to effect a change. We look at module debugging techniques in Chapter 21, "Debugging Device Drivers."

There are also some downsides if you choose to design your driver as a kernel module. Unlike built-in drivers, modules cannot reserve resources during boot time, when success is more or less guaranteed.

Before Starting

Linux has trekked many a terrain and is now state of the art, so you can use it as a vehicle to understand operating system concepts, processor architectures, and even industry domains. When you learn a technique used by a device driver subsystem, look one level deeper and probe the underlying reasons behind that design choice.

Wherever not explicitly stated, the text assumes the 32-bit x86 architecture. The book is, however, mindful of the fact that you are more likely to write device drivers for embedded devices than for conventional PC-compatible systems. The chapter on serial drivers, for example, examines two devices: a touch controller on a PC derivative and a UART on a cell phone. Or the chapter on I^2C device drivers looks at an EEPROM on a PC system and a Real Time Clock on an embedded device. The book also teaches you about the core infrastructure that the kernel provides for most driver classes, which hides architecture dependencies from device drivers.

Device driver debugging techniques are discussed near the end of the book in Chapter 21, so you might find it worthwhile to forward to that chapter as you develop drivers while reading the book.

This book is based on the 2.6 kernel, which has substantial changes across the board from 2.4, touching all major subsystems. Hopefully, you have installed a 2.6-based Linux on your system by now and started experimenting with the kernel sources. Each chapter takes the liberty of profusely pointing you to relevant kernel source files for two main reasons:

1. Because each driver subsystem in the kernel is tens of thousands of lines long, it's only possible to take a relatively simplistic view in a book. Looking at real drivers in the sources along with the example code in this book will give you the bigger picture.

2. Before developing a driver, it's a good idea to zero in on an existing driver in the *drivers/* directory that is similar to your requirement and make that your starting point.

So, to derive maximum benefit from this book, familiarize yourself with the kernel by frequently browsing the source tree and staring hard at the code. And in tandem with your code explorations, follow the goings-on in the kernel mailing list.

Chapter 2. A Peek Inside the Kernel

In This Chapter

- Booting Up 18
- Kernel Mode and User Mode 30
- Process Context and Interrupt Context 30
- Kernel Timers 31
- Concurrency in the Kernel 39
- Process Filesystem 49
- Allocating Memory 49
- Looking at the Sources 52

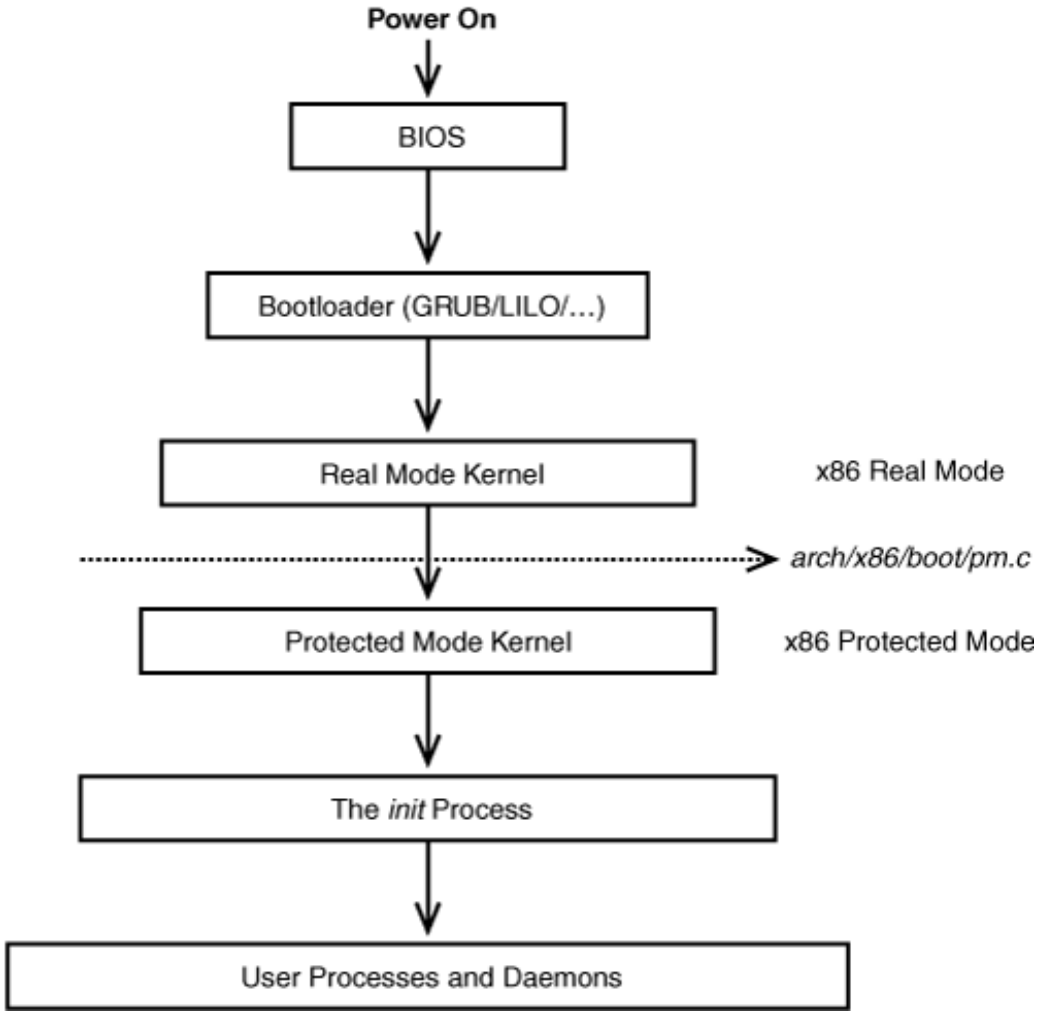
Before we start our journey into the mystical world of Linux device drivers, let's familiarize ourselves with some basic kernel concepts by looking at several kernel regions through the lens of a driver developer. We learn about kernel timers, synchronization mechanisms, and memory allocation. But let's start our expedition by getting a view from the top; let's skim through boot messages emitted by the kernel and hit the breaks whenever something looks interesting.

Booting Up

Figure 2.1 shows the Linux boot sequence on an x86-based computer. Linux boot on x86-based hardware is set into motion when the BIOS loads the *Master Boot Record* (MBR) from the boot device. Code resident in the MBR looks at the partition table and reads a Linux bootloader such as GRUB, LILO, or SYSLINUX from the active

partition. The final stage of the bootloader loads the compressed kernel image and passes control to it. The kernel uncompresses itself and turns on the ignition.

Figure 2.1. Linux boot sequence on x86-based hardware.



x86-based processors have two modes of operation, *real* mode and *protected* mode. In real mode, you can access only the first 1MB of memory, that too without any protection. Protected mode is sophisticated and lets you tap into many advanced features of the processor such as paging. The CPU has to pass through real mode en route to protected mode. This road is a one-way street, however. You can't switch back to real mode from protected mode.

The first-level kernel initializations are done in real mode assembly. Subsequent startup is performed in protected mode by the function `start_kernel()` defined in *init/main.c*, the source file you modified in the previous chapter. `start_kernel()` begins by initializing the CPU subsystem. Memory and process management are put in place soon after. Peripheral buses and I/O devices are started next. As the last step in the boot sequence, the *init* program, the parent of all Linux processes, is invoked. Init executes user-space scripts that start necessary kernel services. It finally spawns terminals on consoles and displays the login prompt.

Each following section header is a message from Figure 2.2 generated during boot progression on an x86-based

laptop. The semantics and the messages may change if you are booting the kernel on other architectures. If some explanations in this section sound rather cryptic, don't worry; the intent here is only to give you a picture from 100 feet above and to let you savor a first taste of the kernel's flavor. Many concepts mentioned here in passing are covered in depth later on.

Figure 2.2. Kernel boot messages.

Code View:

```
Linux version 2.6.23.1y (root@localhost.localdomain) (gcc version 4.1.1 20061011 (Red
Hat 4.1.1-30)) #7 SMP PREEMPT Thu Nov 1 11:39:30 IST 2007
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009f000 (usable)
 BIOS-e820: 000000000009f000 - 00000000000a0000 (reserved)
 ...
758MB LOWMEM available.
...
Kernel command line: ro root=/dev/hda1
...
Console: colour VGA+ 80x25
...
Calibrating delay using timer specific routine.. 1197.46 BogoMIPS (lpj=2394935)
...
CPU: L1 I cache: 32K, L1 D cache: 32K
CPU: L2 cache: 1024K
...
Checking 'hlt' instruction... OK.
...
Setting up standard PCI resources
...
NET: Registered protocol family 2
IP route cache hash table entries: 32768 (order: 5, 131072 bytes)
TCP established hash table entries: 131072 (order: 9, 2097152 bytes)
...
checking if image is initramfs... it is
Freeing initrd memory: 387k freed
...
io scheduler noop registered
io scheduler anticipatory registered (default)
...
00:0a: ttyS0 at I/O 0x3f8 (irq = 4) is a NS16550A
...
Uniform Multi-Platform E-IDE driver Revision: 7.00alpha2
ide: Assuming 33MHz system bus speed for PIO modes; override with idebus=xx
ICH4: IDE controller at PCI slot 0000:00:1f.1
Probing IDE interface ide0...
hda: HTS541010G9AT00, ATA DISK drive
hdc: HL-DT-STCD-RW/DVD DRIVE GCC-4241N, ATAPI CD/DVD-ROM drive
...
serio: i8042 KBD port at 0x60,0x64 irq 1
mice: PS/2 mouse device common for all mice
...
Synaptics Touchpad, model: 1, fw: 5.9, id: 0x2c6ab1, caps: 0x884793/0x0
...
agpgart: Detected an Intel 855GM Chipset.
...
Intel(R) PRO/1000 Network Driver - version 7.3.20-k2
...
ehci_hcd 0000:00:1d.7: EHCI Host Controller
...
```

```
Yenta: CardBus bridge found at 0000:02:00.0 [1014:0560]
...
Non-volatile memory driver v1.2
...
kjournald starting. Commit interval 5 seconds
EXT3 FS on hda2, internal journal
EXT3-fs: mounted filesystem with ordered data mode.
...
INIT: version 2.85 booting
...
```

BIOS-Provided Physical RAM Map

The kernel assembles the system memory map from the BIOS, and this is one of the first boot messages you will see:

```
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 000000000009f000 (usable)
...
BIOS-e820: 00000000ff800000 - 0000000100000000 (reserved)
```

Real mode initialization code uses the BIOS `int 0x15` service with function number `0xe820`(hence the string `BIOS-e820` in the preceding message) to obtain the system memory map. The memory map indicates reserved and usable memory ranges, which is subsequently used by the kernel to create its free memory pool. We discuss more on the BIOS-supplied memory map in the section "Real Mode Calls" in Appendix B, "Linux and the BIOS."

758MB LOWMEM Available

The normally addressable kernel memory region (< 896MB) is called *low memory.* The kernel memory allocator, `kmalloc()`, returns memory from this region. Memory beyond 896MB (called *high memory*) can be accessed only using special mappings.

During boot, the kernel calculates and displays the total pages present in these memory zones. We take a deeper look at memory zones later in this chapter.

Kernel Command Line: ro root=/dev/hda1

Linux bootloaders usually pass a command line to the kernel. Arguments in the command line are similar to the `argv[]` list passed to the `main()` function in C programs, except that they are passed to the kernel instead. You may add command-line arguments to the bootloader configuration file or supply them from the bootloader prompt at runtime.[1] If you are using the GRUB bootloader, the configuration file is either */boot/grub/grub.conf* or */boot/grub/menu.lst* depending on your distribution. If you are a LILO user, the configuration file is */etc/lilo.conf.* An example *grub.conf* file (with comments added) is listed here. You can figure out the genesis of the preceding boot message if you look at the line following `title kernel 2.6.23`:

[1] Bootloaders on embedded devices are usually "slim" and do not support configuration files or equivalent mechanisms. Because of this, many non-x86 architectures support a kernel configuration option called `CONFIG_CMDLINE` that you can use to supply the kernel command line at build time.

```
default 0  #Boot the 2.6.23 kernel by default
```

```
timeout 5  #5 second to alter boot order or parameters

title kernel 2.6.23     #Boot Option 1
  #The boot image resides in the first partition of the first disk
  #under the /boot/ directory and is named vmlinuz-2.6.23. 'ro'
  #indicates that the root partition should be mounted read-only.
  kernel (hd0,0)/boot/vmlinuz-2.6.23 ro root=/dev/hda1

  #Look under section "Freeing initrd memory:387k freed"
  initrd (hd0,0)/boot/initrd

#...
```

Command-line arguments affect the code path traversed during boot. As a simple example, assume that the command-line argument of interest is called `bootmode`. If this parameter is set to `1`, you would like to print some debug messages during boot and switch to a runlevel of `3` at the end of the boot. (Wait until the boot messages are printed out by the init process to learn the semantics of runlevels.) If `bootmode` is instead set to `0`, you would prefer the boot to be relatively laconic, and the runlevel set to `2`. Because you are already familiar with *init/main.c*, let's add the following modification to it:

Code View:

```
static unsigned int bootmode = 1;
static int __init
is_bootmode_setup(char *str)
{
  get_option(&str, &bootmode);
  return 1;
}

/* Handle parameter "bootmode=" */
__setup("bootmode=", is_bootmode_setup);

if (bootmode) {
  /* Print verbose output */
  /* ... */
}

/* ... */

/* If bootmode is 1, choose an init runlevel of 3, else
   switch to a run level of 2 */
if (bootmode) {
  argv_init[++args] = "3";
} else {
  argv_init[++args] = "2";
}

/* ... */
```

Rebuild the kernel as you did earlier and try out the change. We discuss more about kernel command-line arguments in the section "Memory Layout" in Chapter 18, "Embedding Linux."

Calibrating Delay...1197.46 BogoMIPS (lpj=2394935)

During boot, the kernel calculates the number of times the processor can execute an internal delay loop in one *jiffy*, which is the time interval between two consecutive ticks of the system timer. As you would expect, the calculation has to be calibrated to the processing speed of your CPU. The result of this calibration is stored in a kernel variable called `loops_per_jiffy`. One place where the kernel makes use of `loops_per_jiffy` is when a device driver desires to delay execution for small durations in the order of microseconds.

To understand the delay-loop calibration code, let's take a peek inside `calibrate_delay()`, defined in *init/calibrate.c.* This function cleverly derives floating-point precision using the integer kernel. The following snippet (with some comments added) shows the initial portion of the function that carves out a coarse value for `loops_per_jiffy`:

```
loops_per_jiffy = (1 << 12); /* Initial approximation = 4096 */
printk(KERN_DEBUG "Calibrating delay loop... ");
while ((loops_per_jiffy <<= 1) != 0) {
ticks = jiffies;  /* As you will find out in the section, "Kernel
                     Timers," the jiffies variable contains the
                     number of timer ticks since the kernel
                     started, and is incremented in the timer
                     interrupt handler */

  while (ticks == jiffies); /* Wait until the start
                               of the next jiffy */
  ticks = jiffies;
  /* Delay */
  __delay(loops_per_jiffy);

  /* Did the wait outlast the current jiffy? Continue if
     it didn't */
  ticks = jiffies - ticks;
  if (ticks) break;
}

loops_per_jiffy >>= 1; /* This fixes the most significant bit and is
                          the lower-bound of loops_per_jiffy */
```

The preceding code begins by assuming that `loops_per_jiffy` is greater than `4096`, which translates to a processor speed of roughly one *million instructions per second* (MIPS). It then waits for a fresh jiffy to start and executes the delay loop, `__delay(loops_per_jiffy)`. If the delay loop outlasts the jiffy, the previous value of `loops_per_jiffy` (obtained by bitwise right-shifting it by one) fixes its *most significant bit* (MSB). Otherwise, the function continues by checking whether it will obtain the MSB by bitwise left-shifting `loops_per_jiffy`. When the kernel thus figures out the MSB of `loops_per_jiffy`, it works on the lower-order bits and fine-tunes its precision as follows:

```
loopbit = loops_per_jiffy;

/* Gradually work on the lower-order bits */
while (lps_precision-- && (loopbit >>= 1)) {
  loops_per_jiffy |= loopbit;
  ticks = jiffies;
  while (ticks == jiffies); /* Wait until the start
                               of the next jiffy */
ticks = jiffies;
```

```
  /* Delay */
  __delay(loops_per_jiffy);

  if (jiffies != ticks)        /* longer than 1 tick */
    loops_per_jiffy &= ~loopbit;
}
```

The preceding snippet figures out the exact combination of the lower bits of `loops_per_jiffy` when the delay loop crosses a jiffy boundary. This calibrated value is used to derive an unscientific measure of the processor speed, known as *BogoMIPS.* You can use the BogoMIPS rating as a relative measurement of how fast a CPU can run. On a 1.6GHz Pentium M-based laptop, the delay-loop calibration yielded a value of `2394935` for `loops_per_jiffy` as announced by the preceding boot message. The BogoMIPS value is obtained as follows:

BogoMIPS = `loops_per_jiffy` * Number of jiffies in 1 second * Number of instructions consumed by the internal delay loop in units of 1 million

= (2394935 * HZ * 2) / (1 million)

= (2394935 * 250 * 2) / (1000000)

= 1197.46 (as displayed in the preceding boot message)

We further discuss `jiffies`, `HZ`, and `loops_per_jiffy` in the section "Kernel Timers" later in this chapter.

Checking HLT Instruction

Because the Linux kernel is supported on a variety of hardware platforms, the boot code checks for architecture-dependent bugs. Verifying the sanity of the *halt* (HLT) instruction is one such check.

The HLT instruction supported by x86 processors puts the CPU into a low-power sleep mode that continues until the next hardware interrupt occurs. The kernel uses the HLT instruction when it wants to put the CPU in the idle state (see function `cpu_idle()` defined in *arch/x86/kernel/process_32.c*).

For problematic CPUs, the `no-hlt` kernel command-line argument can be used to disable the HLT instruction. If `no-hlt` is turned on, the kernel busy-waits while it's idle, rather than keep the CPU cool by putting it in the HLT state.

The preceding boot message is generated when the startup code in *init/main.c* calls `check_bugs()` defined in *include/asm-your-arch/bugs.h*.

NET: Registered Protocol Family 2

The Linux socket layer is a uniform interface through which user applications access various networking protocols. Each protocol registers itself with the socket layer using a unique family number (defined in *include/linux/socket.h*) assigned to it. Family 2 in the preceding message stands for `AF_INET` (Internet Protocol).

Another registered protocol family often found in boot messages is `AF_NETLINK` (Family 16). Netlink sockets offer a method to communicate between user processes and the kernel. Functionalities accomplished via netlink sockets include accessing the routing table and the *Address Resolution Protocol* (ARP) table (see *include/linux/netlink.h* for the full usage list). Netlink sockets are more suitable than system calls to accomplish such tasks because they are asynchronous, simpler to implement, and dynamically linkable.

Another protocol family commonly enabled in the kernel is `AF_UNIX` or UNIX-domain sockets. Programs such as X Windows use them for interprocess communication on the same system.

Freeing Initrd Memory: 387k Freed

Initrd is a memory-resident virtual disk image loaded by the bootloader. It's mounted as the initial root filesystem after the kernel boots, to hold additional dynamically loadable modules required to mount the disk partition that holds the actual root filesystem. Because the kernel runs on different hardware platforms that use diverse storage controllers, it's not feasible for distributions to enable device drivers for all possible disk drives in the base kernel image. Drivers specific to your system's storage device are packed inside initrd and loaded after the kernel boots, but before the root filesystem is mounted. To create an initrd image, use the `mkinitrd` command.

The 2.6 kernel includes a feature called *initramfs* that bring several benefits over initrd. Whereas the latter emulates a disk (hence called *initramdisk* or *initrd*) and suffers the overheads associated with the Linux block I/O subsystem such as caching, the former essentially gets the cache itself mounted like a filesystem (hence called *initramfs*).

Initramfs, like the page cache over which it's built, grows and shrinks dynamically unlike initrd and, hence, reduces memory wastage. Also, unlike initrd, which requires you to include the associated filesystem driver (e.g., EXT2 drivers if you have an EXT2 filesystem on your initrd), initramfs needs no filesystem support. The initramfs code is tiny because it's just a small layer on top of the page cache.

You can pack your initial root filesystem into a compressed *cpio* archive[2] and pass it to the kernel command line using the `initrd=` argument or build it as part of the kernel image using the `INITRAMFS_SOURCE` menu option during kernel configuration. With the latter, you may either provide the filename of a cpio archive or the path name to a directory tree containing your initramfs layout. During boot, the kernel extracts the files into an initramfs root filesystem (also called *rootfs*) and executes a top-level */init* program if it finds one. This method of obtaining an initial rootfs is especially useful for embedded platforms, where all system resources are at a premium. To create an initramfs image, use `mkinitramfs`. Look at *Documentation/filesystems/ramfs-rootfs-initramfs.txt* for more documentation.

[2] cpio is a UNIX file archival format. You can download it from www.gnu.org/software/cpio.

In this case, we are using initramfs by supplying a compressed cpio archive of the initial root filesystem to the kernel using the `initrd=` command-line argument. After unpacking the contents of the archive into rootfs, the kernel frees the memory where the archive resides (387K in this case) and announces the above boot message. The freed pages are then doled out to other parts of the kernel that request memory.

As discussed in Chapter 18, initrd and initramfs are sometimes used to hold the actual root filesystem on embedded devices during development.

IO Scheduler Anticipatory Registered (Default)

The main goal of an I/O scheduler is to increase system throughput by minimizing disk seek times, which is the latency to move the disk head from its existing position to the disk sector of interest. The 2.6 kernel provides four different I/O schedulers: *Deadline, Anticipatory, Complete Fair Queuing*, and *Noop.* As the preceding kernel message indicates, the kernel sets *Anticipatory* as the default I/O scheduler. We look at I/O scheduling in Chapter 14, "Block Drivers."

Setting Up Standard PCI Resources

The next phase of the boot process probes and initializes I/O buses and peripheral controllers. The kernel probes PCI hardware by walking the PCI bus, and then initializes other I/O subsystems. Take a look at the boot messages in Figure 2.3 to see announcements regarding the initialization of the SCSI subsystem, the USB controller, the video chip (part of the 855 North Bridge chipset in the messages below), the serial port (8250 UART in this case), PS/2 keyboard and mouse, floppy drives, ramdisk, the loopback device, the IDE controller

(part of the ICH4 South Bridge chipset in this example), the touchpad, the Ethernet controller (e1000 in this case), and the PCMCIA controller. The identity of the corresponding I/O device is labeled against →.

Figure 2.3. Initializing buses and peripheral controllers during boot.

Code View:

```
SCSI subsystem initialized                  → SCSI
usbcore: registered new driver hub          → USB
agpgart: Detected an Intel 855 Chipset.     → Video
[drm] Initialized drm 1.0.0 20040925
PS/2 Controller [PNP0303:KBD,PNP0f13:MOU]
at 0x60,0x64 irq 1,12 serio: i8042 KBD port → Keyboard
serial8250: ttyS0 at I/O 0x3f8 (irq = 4)
is a NS16550A                               → Serial Port
Floppy drive(s): fd0 is 1.44M               → Floppy
RAMDISK driver initialized: 16 RAM disks
of 4096K size 1024 blocksize                → Ramdisk
loop: loaded (max 8 devices)                → Loop back
ICH4: IDE controller at PCI slot
0000:00:1f.1                                → Hard Disk
...
input: SynPS/2 Synaptics TouchPad as
/class/input/input1                         → Touchpad
e1000: eth0: e1000_probe: Intel® PRO/1000
Network Connection                          → Ethernet
Yenta: CardBus bridge found at
0000:02:00.0 [1014:0560]                    → PCMCIA/CardBus
...
```

This book discusses many of these driver subsystems in separate chapters. Note that some of these messages might manifest only later on in the boot process if the drivers are dynamically linked to the kernel as loadable modules.

EXT3-fs: Mounted Filesystem

The EXT3 filesystem has become the de facto filesystem on Linux. It adds a journaling layer on top of the veteran EXT2 filesystem to facilitate quick recovery after a crash. The aim is to regain a consistent filesystem state without having to go through a time-consuming filesystem check (`fsck`) operation. EXT2 remains the work engine, while the EXT3 layer additionally logs file transactions on a memory area called *journal* before committing the actual changes to disk. EXT3 is backward-compatible with EXT2, so you can add an EXT3 coating to your existing EXT2 filesystem or peel off the EXT3 to get back your original EXT2 filesystem.

EXT4

The latest version in the EXT filesystem series is EXT4, which has been included in the mainline kernel starting with the 2.6.19 release, with a tag of "experimental" and a name of ext4dev. EXT4 is largely backward-compatible with EXT3. The home page of the EXT4 project is at www.bullopensource.org/ext4.

EXT3 starts a kernel helper thread (we will have an in-depth discussion on kernel threads in the next chapter)

called *kjournald* to assist in journaling. When EXT3 is operational, the kernel mounts the root filesystem and gets ready for business:

```
EXT3-fs: mounted filesystem with ordered data mode
kjournald starting. Commit interval 5 seconds
VFS: Mounted root (ext3 filesystem).
```

INIT: Version 2.85 Booting

Init, the parent of all Linux processes, is the first program to run after the kernel finishes its boot sequence. In the last few lines of *init/main.c*, the kernel searches different locations in its attempt to locate init:

```
if (ramdisk_execute_command) { /* Look for /init in initramfs */
  run_init_process(ramdisk_execute_command);
}

if (execute_command) { /* You may override init and ask the kernel
                          to execute a custom program using the
                          "init=" kernel command-line argument. If
                          you do that, execute_command points to the
                          specified program */
  run_init_process(execute_command);
}

/* Else search for init or sh in the usual places .. */
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");
panic("No init found. Try passing init= option to kernel.");
```

Init receives directions from */etc/inittab*. It first executes system initialization scripts present in */etc/rc.sysinit.* One of the important responsibilities of this script is to activate the swap partition, which triggers a boot message such as this:

```
Adding 1552384k swap on /dev/hda6
```

Let's take a closer look at what this means. Linux user processes own a virtual address space of 3GB (see the section "Allocating Memory"). Out of this, the pages constituting the "working set" are kept in RAM. However, when there are too many programs demanding memory resources, the kernel frees up some used RAM pages by storing them in a disk partition called *swap space.* According to a rule of thumb, the size of the swap partition should be twice the amount of RAM. In this case, the swap space lives in the disk partition */dev/hda6* and has a size of 1552384K bytes.

Init then goes on to run scripts present in the */etc/rc.d/rcX.d/* directory, where *X* is the runlevel specified in inittab. A runlevel is an execution state corresponding to the desired boot mode. For example, multiuser text mode corresponds to a runlevel of 3, while X Windows associates with a runlevel of 5. So, if you see the message, `INIT: Entering runlevel 3`, init has started executing scripts in the */etc/rc.d/rc3.d/* directory. These scripts start the dynamic device-naming subsystem udev (which we discuss in Chapter 4, "Laying the Groundwork") and load kernel modules responsible for driving networking, audio, storage, and so on:

```
Starting udev: [ OK ]
Initializing hardware... network audio storage [Done]
...
```

Init finally spawns terminals on virtual consoles. You can now log in.

Chapter 2. A Peek Inside the Kernel

In This Chapter

- Booting Up 18
- Kernel Mode and User Mode 30
- Process Context and Interrupt Context 30
- Kernel Timers 31
- Concurrency in the Kernel 39
- Process Filesystem 49
- Allocating Memory 49
- Looking at the Sources 52

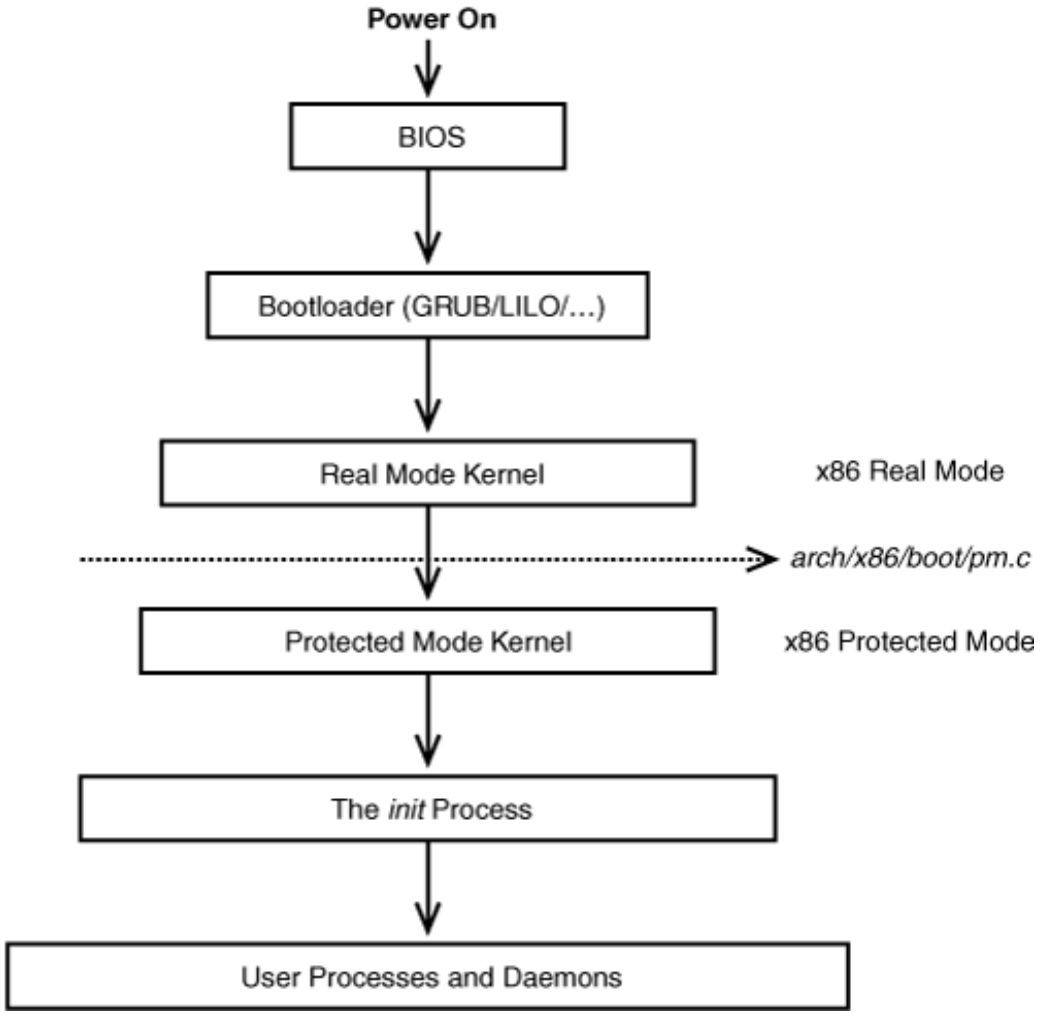
Before we start our journey into the mystical world of Linux device drivers, let's familiarize ourselves with some basic kernel concepts by looking at several kernel regions through the lens of a driver developer. We learn about kernel timers, synchronization mechanisms, and memory allocation. But let's start our expedition by getting a view from the top; let's skim through boot messages emitted by the kernel and hit the breaks whenever something looks interesting.

Booting Up

Figure 2.1 shows the Linux boot sequence on an x86-based computer. Linux boot on x86-based hardware is set into motion when the BIOS loads the *Master Boot Record* (MBR) from the boot device. Code resident in the MBR looks at the partition table and reads a Linux bootloader such as GRUB, LILO, or SYSLINUX from the active

partition. The final stage of the bootloader loads the compressed kernel image and passes control to it. The kernel uncompresses itself and turns on the ignition.

Figure 2.1. Linux boot sequence on x86-based hardware.



x86-based processors have two modes of operation, *real* mode and *protected* mode. In real mode, you can access only the first 1MB of memory, that too without any protection. Protected mode is sophisticated and lets you tap into many advanced features of the processor such as paging. The CPU has to pass through real mode en route to protected mode. This road is a one-way street, however. You can't switch back to real mode from protected mode.

The first-level kernel initializations are done in real mode assembly. Subsequent startup is performed in protected mode by the function `start_kernel()` defined in *init/main.c*, the source file you modified in the previous chapter. `start_kernel()` begins by initializing the CPU subsystem. Memory and process management are put in place soon after. Peripheral buses and I/O devices are started next. As the last step in the boot sequence, the *init* program, the parent of all Linux processes, is invoked. Init executes user-space scripts that start necessary kernel services. It finally spawns terminals on consoles and displays the login prompt.

Each following section header is a message from Figure 2.2 generated during boot progression on an x86-based

laptop. The semantics and the messages may change if you are booting the kernel on other architectures. If some explanations in this section sound rather cryptic, don't worry; the intent here is only to give you a picture from 100 feet above and to let you savor a first taste of the kernel's flavor. Many concepts mentioned here in passing are covered in depth later on.

Figure 2.2. Kernel boot messages.

Code View:

```
Linux version 2.6.23.1y (root@localhost.localdomain) (gcc version 4.1.1 20061011 (Red
Hat 4.1.1-30)) #7 SMP PREEMPT Thu Nov 1 11:39:30 IST 2007
BIOS-provided physical RAM map:
 BIOS-e820: 0000000000000000 - 000000000009f000 (usable)
 BIOS-e820: 000000000009f000 - 00000000000a0000 (reserved)
 ...
758MB LOWMEM available.
...
Kernel command line: ro root=/dev/hda1
...
Console: colour VGA+ 80x25
...
Calibrating delay using timer specific routine.. 1197.46 BogoMIPS (lpj=2394935)
...
CPU: L1 I cache: 32K, L1 D cache: 32K
CPU: L2 cache: 1024K
...
Checking 'hlt' instruction... OK.
...
Setting up standard PCI resources
...
NET: Registered protocol family 2
IP route cache hash table entries: 32768 (order: 5, 131072 bytes)
TCP established hash table entries: 131072 (order: 9, 2097152 bytes)
...
checking if image is initramfs... it is
Freeing initrd memory: 387k freed
...
io scheduler noop registered
io scheduler anticipatory registered (default)
...
00:0a: ttyS0 at I/O 0x3f8 (irq = 4) is a NS16550A
...
Uniform Multi-Platform E-IDE driver Revision: 7.00alpha2
ide: Assuming 33MHz system bus speed for PIO modes; override with idebus=xx
ICH4: IDE controller at PCI slot 0000:00:1f.1
Probing IDE interface ide0...
hda: HTS541010G9AT00, ATA DISK drive
hdc: HL-DT-STCD-RW/DVD DRIVE GCC-4241N, ATAPI CD/DVD-ROM drive
...
serio: i8042 KBD port at 0x60,0x64 irq 1
mice: PS/2 mouse device common for all mice
...
Synaptics Touchpad, model: 1, fw: 5.9, id: 0x2c6ab1, caps: 0x884793/0x0
...
agpgart: Detected an Intel 855GM Chipset.
...
Intel(R) PRO/1000 Network Driver - version 7.3.20-k2
...
ehci_hcd 0000:00:1d.7: EHCI Host Controller
...
```

```
Yenta: CardBus bridge found at 0000:02:00.0 [1014:0560]
...
Non-volatile memory driver v1.2
...
kjournald starting. Commit interval 5 seconds
EXT3 FS on hda2, internal journal
EXT3-fs: mounted filesystem with ordered data mode.
...
INIT: version 2.85 booting
...
```

BIOS-Provided Physical RAM Map

The kernel assembles the system memory map from the BIOS, and this is one of the first boot messages you will see:

```
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 000000000009f000 (usable)
...
BIOS-e820: 00000000ff800000 - 0000000100000000 (reserved)
```

Real mode initialization code uses the BIOS `int 0x15` service with function number `0xe820`(hence the string `BIOS-e820` in the preceding message) to obtain the system memory map. The memory map indicates reserved and usable memory ranges, which is subsequently used by the kernel to create its free memory pool. We discuss more on the BIOS-supplied memory map in the section "Real Mode Calls" in Appendix B, "Linux and the BIOS."

758MB LOWMEM Available

The normally addressable kernel memory region (< 896MB) is called *low memory.* The kernel memory allocator, `kmalloc()`, returns memory from this region. Memory beyond 896MB (called *high memory*) can be accessed only using special mappings.

During boot, the kernel calculates and displays the total pages present in these memory zones. We take a deeper look at memory zones later in this chapter.

Kernel Command Line: ro root=/dev/hda1

Linux bootloaders usually pass a command line to the kernel. Arguments in the command line are similar to the `argv[]` list passed to the `main()` function in C programs, except that they are passed to the kernel instead. You may add command-line arguments to the bootloader configuration file or supply them from the bootloader prompt at runtime.[1] If you are using the GRUB bootloader, the configuration file is either */boot/grub/grub.conf* or */boot/grub/menu.lst* depending on your distribution. If you are a LILO user, the configuration file is */etc/lilo.conf.* An example *grub.conf* file (with comments added) is listed here. You can figure out the genesis of the preceding boot message if you look at the line following `title kernel 2.6.23`:

[1] Bootloaders on embedded devices are usually "slim" and do not support configuration files or equivalent mechanisms. Because of this, many non-x86 architectures support a kernel configuration option called `CONFIG_CMDLINE` that you can use to supply the kernel command line at build time.

```
default 0  #Boot the 2.6.23 kernel by default
```

```
timeout 5  #5 second to alter boot order or parameters

title kernel 2.6.23     #Boot Option 1
  #The boot image resides in the first partition of the first disk
  #under the /boot/ directory and is named vmlinuz-2.6.23. 'ro'
  #indicates that the root partition should be mounted read-only.
  kernel (hd0,0)/boot/vmlinuz-2.6.23 ro root=/dev/hda1

  #Look under section "Freeing initrd memory:387k freed"
  initrd (hd0,0)/boot/initrd

#...
```

Command-line arguments affect the code path traversed during boot. As a simple example, assume that the command-line argument of interest is called `bootmode`. If this parameter is set to `1`, you would like to print some debug messages during boot and switch to a runlevel of `3` at the end of the boot. (Wait until the boot messages are printed out by the init process to learn the semantics of runlevels.) If `bootmode` is instead set to `0`, you would prefer the boot to be relatively laconic, and the runlevel set to `2`. Because you are already familiar with *init/main.c*, let's add the following modification to it:

Code View:

```
static unsigned int bootmode = 1;
static int __init
is_bootmode_setup(char *str)
{
  get_option(&str, &bootmode);
  return 1;
}

/* Handle parameter "bootmode=" */
__setup("bootmode=", is_bootmode_setup);

if (bootmode) {
  /* Print verbose output */
  /* ... */
}

/* ... */

/* If bootmode is 1, choose an init runlevel of 3, else
   switch to a run level of 2 */
if (bootmode) {
  argv_init[++args] = "3";
} else {
  argv_init[++args] = "2";
}

/* ... */
```

Rebuild the kernel as you did earlier and try out the change. We discuss more about kernel command-line arguments in the section "Memory Layout" in Chapter 18, "Embedding Linux."

Calibrating Delay...1197.46 BogoMIPS (lpj=2394935)

During boot, the kernel calculates the number of times the processor can execute an internal delay loop in one *jiffy*, which is the time interval between two consecutive ticks of the system timer. As you would expect, the calculation has to be calibrated to the processing speed of your CPU. The result of this calibration is stored in a kernel variable called `loops_per_jiffy`. One place where the kernel makes use of `loops_per_jiffy` is when a device driver desires to delay execution for small durations in the order of microseconds.

To understand the delay-loop calibration code, let's take a peek inside `calibrate_delay()`, defined in *init/calibrate.c.* This function cleverly derives floating-point precision using the integer kernel. The following snippet (with some comments added) shows the initial portion of the function that carves out a coarse value for `loops_per_jiffy`:

```
loops_per_jiffy = (1 << 12); /* Initial approximation = 4096 */
printk(KERN_DEBUG "Calibrating delay loop... ");
while ((loops_per_jiffy <<= 1) != 0) {
ticks = jiffies;  /* As you will find out in the section, "Kernel
                     Timers," the jiffies variable contains the
                     number of timer ticks since the kernel
                     started, and is incremented in the timer
                     interrupt handler */

  while (ticks == jiffies); /* Wait until the start
                               of the next jiffy */
  ticks = jiffies;
  /* Delay */
  __delay(loops_per_jiffy);

  /* Did the wait outlast the current jiffy? Continue if
     it didn't */
  ticks = jiffies - ticks;
  if (ticks) break;
}

loops_per_jiffy >>= 1; /* This fixes the most significant bit and is
                          the lower-bound of loops_per_jiffy */
```

The preceding code begins by assuming that `loops_per_jiffy` is greater than `4096`, which translates to a processor speed of roughly one *million instructions per second* (MIPS). It then waits for a fresh jiffy to start and executes the delay loop, `__delay(loops_per_jiffy)`. If the delay loop outlasts the jiffy, the previous value of `loops_per_jiffy` (obtained by bitwise right-shifting it by one) fixes its *most significant bit* (MSB). Otherwise, the function continues by checking whether it will obtain the MSB by bitwise left-shifting `loops_per_jiffy`. When the kernel thus figures out the MSB of `loops_per_jiffy`, it works on the lower-order bits and fine-tunes its precision as follows:

```
loopbit = loops_per_jiffy;

/* Gradually work on the lower-order bits */
while (lps_precision-- && (loopbit >>= 1)) {
  loops_per_jiffy |= loopbit;
  ticks = jiffies;
  while (ticks == jiffies); /* Wait until the start
                               of the next jiffy */
ticks = jiffies;
```

```
  /* Delay */
  __delay(loops_per_jiffy);

  if (jiffies != ticks)        /* longer than 1 tick */
    loops_per_jiffy &= ~loopbit;
}
```

The preceding snippet figures out the exact combination of the lower bits of `loops_per_jiffy` when the delay loop crosses a jiffy boundary. This calibrated value is used to derive an unscientific measure of the processor speed, known as *BogoMIPS.* You can use the BogoMIPS rating as a relative measurement of how fast a CPU can run. On a 1.6GHz Pentium M-based laptop, the delay-loop calibration yielded a value of `2394935` for `loops_per_jiffy` as announced by the preceding boot message. The BogoMIPS value is obtained as follows:

BogoMIPS = `loops_per_jiffy` * Number of jiffies in 1 second * Number of instructions consumed by the internal delay loop in units of 1 million

= (2394935 * HZ * 2) / (1 million)

= (2394935 * 250 * 2) / (1000000)

= 1197.46 (as displayed in the preceding boot message)

We further discuss `jiffies`, `HZ`, and `loops_per_jiffy` in the section "Kernel Timers" later in this chapter.

Checking HLT Instruction

Because the Linux kernel is supported on a variety of hardware platforms, the boot code checks for architecture-dependent bugs. Verifying the sanity of the *halt* (HLT) instruction is one such check.

The HLT instruction supported by x86 processors puts the CPU into a low-power sleep mode that continues until the next hardware interrupt occurs. The kernel uses the HLT instruction when it wants to put the CPU in the idle state (see function `cpu_idle()` defined in *arch/x86/kernel/process_32.c*).

For problematic CPUs, the `no-hlt` kernel command-line argument can be used to disable the HLT instruction. If `no-hlt` is turned on, the kernel busy-waits while it's idle, rather than keep the CPU cool by putting it in the HLT state.

The preceding boot message is generated when the startup code in *init/main.c* calls `check_bugs()` defined in *include/asm-your-arch/bugs.h*.

NET: Registered Protocol Family 2

The Linux socket layer is a uniform interface through which user applications access various networking protocols. Each protocol registers itself with the socket layer using a unique family number (defined in *include/linux/socket.h*) assigned to it. Family 2 in the preceding message stands for `AF_INET` (Internet Protocol).

Another registered protocol family often found in boot messages is `AF_NETLINK` (Family 16). Netlink sockets offer a method to communicate between user processes and the kernel. Functionalities accomplished via netlink sockets include accessing the routing table and the *Address Resolution Protocol* (ARP) table (see *include/linux/netlink.h* for the full usage list). Netlink sockets are more suitable than system calls to accomplish such tasks because they are asynchronous, simpler to implement, and dynamically linkable.

Another protocol family commonly enabled in the kernel is `AF_UNIX` or UNIX-domain sockets. Programs such as X Windows use them for interprocess communication on the same system.

Freeing Initrd Memory: 387k Freed

Initrd is a memory-resident virtual disk image loaded by the bootloader. It's mounted as the initial root filesystem after the kernel boots, to hold additional dynamically loadable modules required to mount the disk partition that holds the actual root filesystem. Because the kernel runs on different hardware platforms that use diverse storage controllers, it's not feasible for distributions to enable device drivers for all possible disk drives in the base kernel image. Drivers specific to your system's storage device are packed inside initrd and loaded after the kernel boots, but before the root filesystem is mounted. To create an initrd image, use the `mkinitrd` command.

The 2.6 kernel includes a feature called *initramfs* that bring several benefits over initrd. Whereas the latter emulates a disk (hence called *initramdisk* or *initrd*) and suffers the overheads associated with the Linux block I/O subsystem such as caching, the former essentially gets the cache itself mounted like a filesystem (hence called *initramfs*).

Initramfs, like the page cache over which it's built, grows and shrinks dynamically unlike initrd and, hence, reduces memory wastage. Also, unlike initrd, which requires you to include the associated filesystem driver (e.g., EXT2 drivers if you have an EXT2 filesystem on your initrd), initramfs needs no filesystem support. The initramfs code is tiny because it's just a small layer on top of the page cache.

You can pack your initial root filesystem into a compressed *cpio* archive[2] and pass it to the kernel command line using the `initrd=` argument or build it as part of the kernel image using the `INITRAMFS_SOURCE` menu option during kernel configuration. With the latter, you may either provide the filename of a cpio archive or the path name to a directory tree containing your initramfs layout. During boot, the kernel extracts the files into an initramfs root filesystem (also called *rootfs*) and executes a top-level */init* program if it finds one. This method of obtaining an initial rootfs is especially useful for embedded platforms, where all system resources are at a premium. To create an initramfs image, use `mkinitramfs`. Look at *Documentation/filesystems/ramfs-rootfs-initramfs.txt* for more documentation.

[2] cpio is a UNIX file archival format. You can download it from www.gnu.org/software/cpio.

In this case, we are using initramfs by supplying a compressed cpio archive of the initial root filesystem to the kernel using the `initrd=` command-line argument. After unpacking the contents of the archive into rootfs, the kernel frees the memory where the archive resides (387K in this case) and announces the above boot message. The freed pages are then doled out to other parts of the kernel that request memory.

As discussed in Chapter 18, initrd and initramfs are sometimes used to hold the actual root filesystem on embedded devices during development.

IO Scheduler Anticipatory Registered (Default)

The main goal of an I/O scheduler is to increase system throughput by minimizing disk seek times, which is the latency to move the disk head from its existing position to the disk sector of interest. The 2.6 kernel provides four different I/O schedulers: *Deadline, Anticipatory, Complete Fair Queuing*, and *Noop.* As the preceding kernel message indicates, the kernel sets *Anticipatory* as the default I/O scheduler. We look at I/O scheduling in Chapter 14, "Block Drivers."

Setting Up Standard PCI Resources

The next phase of the boot process probes and initializes I/O buses and peripheral controllers. The kernel probes PCI hardware by walking the PCI bus, and then initializes other I/O subsystems. Take a look at the boot messages in Figure 2.3 to see announcements regarding the initialization of the SCSI subsystem, the USB controller, the video chip (part of the 855 North Bridge chipset in the messages below), the serial port (8250 UART in this case), PS/2 keyboard and mouse, floppy drives, ramdisk, the loopback device, the IDE controller

(part of the ICH4 South Bridge chipset in this example), the touchpad, the Ethernet controller (e1000 in this case), and the PCMCIA controller. The identity of the corresponding I/O device is labeled against →.

Figure 2.3. Initializing buses and peripheral controllers during boot.

Code View:

```
SCSI subsystem initialized                  → SCSI
usbcore: registered new driver hub          → USB
agpgart: Detected an Intel 855 Chipset.     → Video
[drm] Initialized drm 1.0.0 20040925
PS/2 Controller [PNP0303:KBD,PNP0f13:MOU]
at 0x60,0x64 irq 1,12 serio: i8042 KBD port → Keyboard
serial8250: ttyS0 at I/O 0x3f8 (irq = 4)
is a NS16550A                               → Serial Port
Floppy drive(s): fd0 is 1.44M               → Floppy
RAMDISK driver initialized: 16 RAM disks
of 4096K size 1024 blocksize                → Ramdisk
loop: loaded (max 8 devices)                → Loop back
ICH4: IDE controller at PCI slot
0000:00:1f.1                                → Hard Disk
...
input: SynPS/2 Synaptics TouchPad as
/class/input/input1                         → Touchpad
e1000: eth0: e1000_probe: Intel® PRO/1000
Network Connection                          → Ethernet
Yenta: CardBus bridge found at
0000:02:00.0 [1014:0560]                    → PCMCIA/CardBus
...
```

This book discusses many of these driver subsystems in separate chapters. Note that some of these messages might manifest only later on in the boot process if the drivers are dynamically linked to the kernel as loadable modules.

EXT3-fs: Mounted Filesystem

The EXT3 filesystem has become the de facto filesystem on Linux. It adds a journaling layer on top of the veteran EXT2 filesystem to facilitate quick recovery after a crash. The aim is to regain a consistent filesystem state without having to go through a time-consuming filesystem check (`fsck`) operation. EXT2 remains the work engine, while the EXT3 layer additionally logs file transactions on a memory area called *journal* before committing the actual changes to disk. EXT3 is backward-compatible with EXT2, so you can add an EXT3 coating to your existing EXT2 filesystem or peel off the EXT3 to get back your original EXT2 filesystem.

EXT4

The latest version in the EXT filesystem series is EXT4, which has been included in the mainline kernel starting with the 2.6.19 release, with a tag of "experimental" and a name of ext4dev. EXT4 is largely backward-compatible with EXT3. The home page of the EXT4 project is at www.bullopensource.org/ext4.

EXT3 starts a kernel helper thread (we will have an in-depth discussion on kernel threads in the next chapter)

called *kjournald* to assist in journaling. When EXT3 is operational, the kernel mounts the root filesystem and gets ready for business:

```
EXT3-fs: mounted filesystem with ordered data mode
kjournald starting. Commit interval 5 seconds
VFS: Mounted root (ext3 filesystem).
```

INIT: Version 2.85 Booting

Init, the parent of all Linux processes, is the first program to run after the kernel finishes its boot sequence. In the last few lines of *init/main.c*, the kernel searches different locations in its attempt to locate init:

```
if (ramdisk_execute_command) { /* Look for /init in initramfs */
  run_init_process(ramdisk_execute_command);
}

if (execute_command) { /* You may override init and ask the kernel
                          to execute a custom program using the
                          "init=" kernel command-line argument. If
                          you do that, execute_command points to the
                          specified program */
  run_init_process(execute_command);
}

/* Else search for init or sh in the usual places .. */
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");
panic("No init found. Try passing init= option to kernel.");
```

Init receives directions from */etc/inittab*. It first executes system initialization scripts present in */etc/rc.sysinit.* One of the important responsibilities of this script is to activate the swap partition, which triggers a boot message such as this:

```
Adding 1552384k swap on /dev/hda6
```

Let's take a closer look at what this means. Linux user processes own a virtual address space of 3GB (see the section "Allocating Memory"). Out of this, the pages constituting the "working set" are kept in RAM. However, when there are too many programs demanding memory resources, the kernel frees up some used RAM pages by storing them in a disk partition called *swap space.* According to a rule of thumb, the size of the swap partition should be twice the amount of RAM. In this case, the swap space lives in the disk partition */dev/hda6* and has a size of 1552384K bytes.

Init then goes on to run scripts present in the */etc/rc.d/rcX.d/* directory, where *X* is the runlevel specified in inittab. A runlevel is an execution state corresponding to the desired boot mode. For example, multiuser text mode corresponds to a runlevel of 3, while X Windows associates with a runlevel of 5. So, if you see the message, `INIT: Entering runlevel 3`, init has started executing scripts in the */etc/rc.d/rc3.d/* directory. These scripts start the dynamic device-naming subsystem udev (which we discuss in Chapter 4, "Laying the Groundwork") and load kernel modules responsible for driving networking, audio, storage, and so on:

```
Starting udev: [ OK ]
Initializing hardware... network audio storage [Done]
...
```

Init finally spawns terminals on virtual consoles. You can now log in.

Kernel Mode and User Mode

Some operating systems, such as MS-DOS, always execute in a single CPU mode, but UNIX-like operating systems use dual modes to effectively implement time-sharing. On a Linux machine, the CPU is either in a trusted *kernel mode* or in a restricted *user mode.* All user processes execute in user mode, whereas the kernel itself executes in kernel mode.

Kernel mode code has unrestricted access to the entire processor instruction set and to the full memory and I/O space. If a user mode process needs these privileges, it has to channel requests through device drivers or other kernel mode code via system calls. User mode code is allowed to page fault, however, whereas kernel mode code isn't.

In 2.4 and earlier kernels, only user mode processes could be context switched out and replaced by other processes. Kernel mode code could monopolize the processor until either

- It voluntarily relinquished the CPU.

or

- An interrupt or an exception occurred.

With the introduction of kernel preemption in the 2.6 release, most kernel mode code can also be context switched.

Process Context and Interrupt Context

The kernel accomplishes useful work using a combination of process contexts and interrupt contexts. Kernel code that services system calls issued by user applications runs on behalf of the corresponding application processes and is said to execute in process context. Interrupt handlers, on the other hand, run asynchronously in interrupt context. Processes contexts are not tied to any interrupt context and vice versa.

Kernel code running in process context is preemptible. An interrupt context, however, always runs to completion and is not preemptible. Because of this, there are restrictions on what can be done from interrupt context. Code executing from interrupt context cannot do the following:

- Go to sleep or relinquish the processor
- Acquire a mutex
- Perform time-consuming tasks
- Access user space virtual memory

Look at section "Interrupt Handing" in Chapter 4 for a full discussion of the interrupt context.

Kernel Timers

The working of many parts of the kernel is critically dependent on the passage of time. The Linux kernel makes use of different timers provided by the hardware to provide time-dependent services such as busy-waiting and sleep-waiting. The processor wastes cycles while it busy-waits but relinquishes the CPU when it sleep-waits. Naturally, the former is done only when the latter is not feasible. The kernel also facilitates scheduling of functions that desire to run after a specified time duration has elapsed.

Let's first discuss the semantics of some important kernel timer variables such as `jiffies`, `HZ`, and `xtime`. Next, let's measure execution times on a Pentium-based system using the Pentium *Time Stamp Counter* (TSC). Let's also see how Linux uses the *Real Time Clock* (RTC).

HZ and Jiffies

System timers interrupt the processor (or "pop") at programmable frequencies. This frequency, or the number of timer ticks per second, is contained in the kernel variable `HZ`. Choosing a value for `HZ` is a trade-off. A large `HZ` results in finer timer granularity, and hence better scheduling resolution. However, bigger values of `HZ` also result in larger overhead and higher power consumption, because more cycles are burnt in the timer interrupt context.

> The value of `HZ` is architecture-dependent. On x86 systems, `HZ` used to be set to 100 in 2.4 kernels by default. With 2.6, this value changed to 1000, but with 2.6.13, it was lowered to 250. On ARM-based platforms, 2.6 kernels set `HZ` to 100. With current kernels, you can choose a value for `HZ` at build time through the configuration menu. The default setting for this option depends on your distribution.
>
> The 2.6.21 kernel introduced support for a tickless kernel (`CONFIG_NO_HZ`), which dynamically triggers timer interrupts depending on system load. The tickless system implementation is outside the scope of this chapter.

`jiffies` holds the number of times the system timer has popped since the system booted. The kernel increments the `jiffies` variable, `HZ` times every second. Thus, on a kernel with a `HZ` value of 100, a jiffy is a 10-millisecond duration, whereas on a kernel with `HZ` set to 1000, a jiffy is only 1-millisecond long.

To better understand `HZ` and `jiffies`, consider the following code snippet from the IDE driver (*drivers/ide/ide.c*) that polls disk drives for busy status:

```
unsigned long timeout = jiffies + (3*HZ);
while (hwgroup->busy) {
  /* ... */
  if (time_after(jiffies, timeout)) {
    return -EBUSY;
  }
  /* ... */
}
return SUCCESS;
```

The preceding code returns `SUCCESS` if the busy condition gets cleared in less than 3 seconds, and `-EBUSY` otherwise. `3*HZ` is the number of `jiffies` present in 3 seconds. The calculated timeout, (`jiffies + 3*HZ`), is thus the new value of `jiffies` after the 3-second timeout elapses. The `time_after()` macro compares the current value of `jiffies` with the requested timeout, taking care to account for wraparound due to overflows. Related functions available for doing similar comparisons are `time_before()`, `time_before_eq()`, and `time_after_eq()`.

`jiffies` is defined as *volatile*, which asks the compiler not to optimize access to the variable. This ensures that `jiffies`, which is updated by the timer interrupt handler during each tick, is reread during each pass through the loop.

For the reverse conversion from `jiffies` to seconds, take a look at this snippet from the USB host controller driver, *drivers/usb/host/ehci-sched.c*:

```
if (stream->rescheduled) {
  ehci_info(ehci, "ep%ds-iso rescheduled " "%lu times in %lu
            seconds\n", stream->bEndpointAddress, is_in? "in":
            "out", stream->rescheduled,
            ((jiffies – stream->start)/HZ));
}
```

The preceding debug statement calculates the amount of time in seconds within which this USB endpoint stream (we discuss USB in Chapter 11, "Universal Serial Bus") was rescheduled `stream->rescheduled` times. (`jiffies-stream->start`) is the number of `jiffies` that elapsed since the rescheduling started. The division by `HZ` converts that value into seconds.

The 32-bit `jiffies` variable overflows in approximately 50 days, assuming a `HZ` value of 1000. Because system uptimes can be many times that duration, the kernel provides a variable called `jiffies_64` to hold 64-bit (`u64`) `jiffies`. The linker positions `jiffies_64` such that its bottom 32 bits collocate with `jiffies`. On 32-bit machines, the compiler needs two instructions to assign one `u64` variable to another, so reading `jiffies_64` is not atomic. To get around this problem, the kernel provides a function, `get_jiffies_64()`. Look at `cpufreq_stats_update()` defined in *drivers/cpufreq/cpufreq_stats.c* for a usage example.

Long Delays

In kernel terms, delays in the order of `jiffies` are considered long durations. A possible, but nonoptimal, way to accomplish long delays is by busy-looping. A function that busy-waits has a dog-in-the-manger attitude. It neither uses the processor for doing useful work nor lets others use it. The following code hogs the processor for 1 second:

```
unsigned long timeout = jiffies + HZ;
while (time_before(jiffies, timeout)) continue;
```

A better approach is to sleep-wait, instead of busy-wait. Your code yields the processor to others, while waiting for the time delay to elapse. This is done using `schedule_timeout()`:

```
unsigned long timeout = jiffies + HZ;
schedule_timeout(timeout);  /* Allow other parts of the
                               kernel to run */
```

The delay guarantee is only on the lower bound of the timeout. Whether from kernel space or from user space, it's difficult to get more precise control over timeouts than the granularity of `HZ` because process time slices are updated by the kernel scheduler only during timer ticks. Also, even if your process is scheduled to run after the

specified timeout, the scheduler can decide to pick another process from the run queue based on priorities.[3]

[3] These scheduler properties have changed with the advent of the CFS scheduler in the 2.6.23 kernel. Linux process schedulers are discussed in Chapter 19, "Drivers in User Space."

Two other functions that facilitate sleep-waiting are `wait_event_timeout()` and `msleep()`. Both of them are implemented with the help of `schedule_timeout()`. `wait_event_timeout()` is used when your code desires to resume execution if a specified condition becomes true or if a timeout occurs. `msleep()` sleeps for the specified number of milliseconds.

Such long-delay techniques are suitable for use only from process context. Sleep-waiting cannot be done from interrupt context because interrupt handlers are not allowed to `schedule()` or sleep. (See "Interrupt Handling" in Chapter 4 for a list of do's and don'ts for code executing in interrupt context.) Busy-waiting for a short duration is possible from interrupt context, but long busy-waiting in that context is considered a mortal sin. Equally taboo is long busy-waiting with interrupts disabled.

The kernel also provides timer APIs to execute a function at a point of time in the future. You can dynamically define a timer using `init_timer()` or statically create one with `DEFINE_TIMER()`. After this is done, populate a `timer_list` with the address and parameters of your handler function, and register it using `add_timer()`:

```
#include <linux/timer.h>

struct timer_list my_timer;

init_timer(&my_timer);            /* Also see setup_timer() */
my_timer.expire = jiffies + n*HZ; /* n is the timeout in number
                                     of seconds */
my_timer.function = timer_func;   /* Function to execute
                                     after n seconds */
my_timer.data = func_parameter;   /* Parameter to be passed
                                     to timer_func */
add_timer(&my_timer);             /* Start the timer */
```

Note that this is a one-shot timer. If you want to run `timer_func()` periodically, you also need to add the preceding code inside `timer_func()` to schedule itself after the next timeout:

```
static void timer_func(unsigned long func_parameter)
{
  /* Do work to be done periodically */
  /* ... */

  init_timer(&my_timer);
  my_timer.expire   = jiffies + n*HZ;
  my_timer.data     = func_parameter;
  my_timer.function = timer_func;
  add_timer(&my_timer);
}
```

You may use `mod_timer()` to change the expiration of `my_timer`, `del_timer()` to cancel `my_timer`, and `timer_pending()` to see whether `my_timer` is pending at the moment. If you look at *kernel/timer.c*, you will find that `schedule_timeout()` internally uses these same APIs.

User-space functions such as `clock_settime()` and `clock_gettime()` are used to access kernel timer services from user space. A user application may use `setitimer()` and `getitimer()` to control the delivery of an alarm signal when a specified timeout expires.

Short Delays

In kernel terms, sub-jiffy delays qualify as short durations. Such delays are commonly requested from both process and interrupt contexts. Because it is not possible to use jiffy-based methods to implement sub-jiffy delays, the methods discussed in the previous section to sleep-wait cannot be used for small timeouts. The only solution is to busy-wait.

Kernel APIs that implement short delays are `mdelay()`, `udelay()`, and `ndelay()`, which support millisecond, microsecond, and nanosecond delays, respectively. The actual implementations of these functions are architecture-specific and may not be available on all platforms.

Busy-waiting for short durations is accomplished by measuring the time the processor takes to execute an instruction and looping for the necessary number of iterations. As discussed earlier in this chapter, the kernel performs this measurement during boot and stores the value in a variable called `loops_per_jiffy`. The short-delay APIs use `loops_per_jiffy` to decide the number of times they need to busy-loop. To achieve a 1-microsecond delay during a handshake process, the USB host controller driver, *drivers/usb/host/ehci-hcd.c*, calls `udelay()`, which internally uses `loops_per_jiffy`:

```
do {
  result = ehci_readl(ehci, ptr);
  /* ... */
  if (result == done) return 0;
  udelay(1);     /* Internally uses loops_per_jiffy */
  usec--;
} while (usec > 0);
```

Pentium Time Stamp Counter

The *Time Stamp Counter* (TSC) is a 64-bit register present in Pentium-compatible processors that counts the number of clock cycles consumed by the processor since startup. Because the TSC gets incremented at the rate of the processor cycle speed, it provides a high-resolution timer. The TSC is commonly used for profiling and instrumenting code. It is accessed using the `rdtsc` instruction to measure execution time of intervening code with microsecond precision. TSC ticks can be converted to seconds by dividing by the CPU clock speed, which can be read from the kernel variable, `cpu_khz`.

In the following snippet, `low_tsc_ticks` contains the lower 32 bits of the TSC, while `high_tsc_ticks` contains the higher 32 bits. The lower 32 bits overflow in a few seconds depending on your processor speed but is sufficient for many code instrumentation purposes as shown here:

```
unsigned long low_tsc_ticks0, high_tsc_ticks0;
unsigned long low_tsc_ticks1, high_tsc_ticks1;
unsigned long exec_time;
rdtsc(low_tsc_ticks0, high_tsc_ticks0); /* Timestamp
                                           before */
printk("Hello World\n");                /* Code to be
                                           profiled */
rdtsc(low_tsc_ticks1, high_tsc_ticks1); /* Timestamp after */
exec_time = low_tsc_ticks1 - low_tsc_ticks0;
```

`exec_time` measured 871 (or half a microsecond) on a 1.8GHz Pentium box.

Support for high-resolution timers (`CONFIG_HIGH_RES_TIMERS`) has been merged with the 2.6.21 kernel. It makes use of hardware-specific high-speed timers to provide high-precision capabilities to APIs such as `nanosleep()`. On Pentium-class machines, the kernel leverages the TSC to offer this capability.

Real Time Clock

The RTC tracks absolute time in nonvolatile memory. On x86 PCs, RTC registers constitute the top few locations of a small chunk of battery-powered[4] *complementary metal oxide semiconductor* (CMOS) memory. Look at Figure 5.1 in Chapter 5, "Character Drivers," for the location of the CMOS in the legacy PC architecture. On embedded systems, the RTC might be internal to the processor, or externally connected via the I^2C or SPI buses discussed in Chapter 8, "The Inter-Integrated Circuit Protocol."

[4] RTC batteries last for many years and usually outlive the life span of computers, so you should never have to replace them.

You may use the RTC to do the following:

- Read and set the absolute clock, and generate interrupts during clock updates.
- Generate periodic interrupts with frequencies ranging from 2HZ to 8192HZ.
- Set alarms

Many applications need the concept of absolute time or *wall time.* Because `jiffies` is relative to the time when the system booted, it does not contain wall time. The kernel maintains wall time in a variable called `xtime`. During boot, `xtime` is initialized to the current wall time by reading the RTC. When the system halts, the wall time is written back to the RTC. You can use `do_gettimeofday()` to read wall time with the highest resolution supported by the hardware:

```
#include <linux/time.h>
static struct timeval curr_time;
do_gettimeofday(&curr_time);
my_timestamp = cpu_to_le32(curr_time.tv_sec); /* Record timestamp */
```

There are also a bunch of functions available to user-space code to access wall time. They include the following:

- `time()`, which returns the calendar time, or the number of seconds since Epoch (00:00:00 on January 1, 1970)
- `localtime()`, which returns the calendar time in broken-down format
- `mktime()`, which does the reverse of `localtime()`

- `gettimeofday()`, which returns the calendar time with microsecond precision if your platform supports it

Another way to use the RTC from user space is via the character device, */dev/rtc.* Only one process is allowed to access this device at a time.

We discuss more about RTC drivers in Chapter 5 and Chapter 8. In Chapter 19, we develop an example user application that uses */dev/rtc* to perform periodic work with microsecond precision.

Concurrency in the Kernel

With the arrival of multicore laptops, *Symmetric Multi Processing* (SMP) is no longer confined to the realm of hi-tech users. SMP and kernel preemption are scenarios that generate multiple threads of execution. These threads can simultaneously operate on shared kernel data structures. Because of this, accesses to such data structures have to be serialized.

Let's discuss the basics of protecting shared kernel resources from concurrent access. We start with a simple example and gradually introduce complexities such as interrupts, kernel preemption, and SMP.

Spinlocks and Mutexes

A code area that accesses shared resources is called a *critical* section. Spinlocks and mutexes (short for *mutual exclusion*) are the two basic mechanisms used to protect critical sections in the kernel. Let's look at each in turn.

A spinlock ensures that only a single thread enters a critical section at a time. Any other thread that desires to enter the critical section has to remain spinning at the door until the first thread exits. Note that we use the term *thread* to refer to a thread of execution, rather than a kernel thread.

The basic usage of spinlocks is as follows:

```
#include <linux/spinlock.h>
spinlock_t mylock = SPIN_LOCK_UNLOCKED; /* Initialize */

/* Acquire the spinlock. This is inexpensive if there
 * is no one inside the critical section. In the face of
 * contention, spinlock() has to busy-wait.
 */
spin_lock(&mylock);

/* ... Critical Section code ... */

spin_unlock(&mylock); /* Release the lock */
```

In contrast to spinlocks that put threads into a spin if they attempt to enter a busy critical section, mutexes put contending threads to sleep until it's their turn to occupy the critical section. Because it's a bad thing to consume processor cycles to spin, mutexes are more suitable than spinlocks to protect critical sections when the estimated wait time is long. In mutex terms, anything more than two context switches is considered long, because a mutex has to switch out the contending thread to sleep, and switch it back in when it's time to wake it up.

In many cases, therefore, it's easy to decide whether to use a spinlock or a mutex:

- If the critical section needs to sleep, you have no choice but to use a mutex. It's illegal to schedule, preempt, or sleep on a wait queue after acquiring a spinlock.

- Because mutexes put the calling thread to sleep in the face of contention, you have no choice but to use spinlocks inside interrupt handlers. (You will learn more about the constraints of the interrupt context in Chapter 4.)

Basic mutex usage is as follows:

```
#include <linux/mutex.h>

/* Statically declare a mutex. To dynamically
   create a mutex, use mutex_init() */
static DEFINE_MUTEX(mymutex);

/* Acquire the mutex. This is inexpensive if there
 * is no one inside the critical section. In the face of
 * contention, mutex_lock() puts the calling thread to sleep.
 */
mutex_lock(&mymutex);

/* ... Critical Section code ... */

mutex_unlock(&mymutex);      /* Release the mutex */
```

To illustrate the use of concurrency protection, let's start with a critical section that is present only in process context and gradually introduce complexities in the following order:

1. Critical section present only in process context on a Uniprocessor (UP) box running a nonpreemptible kernel.

2. Critical section present in process and interrupt contexts on a UP machine running a nonpreemptible kernel.

3. Critical section present in process and interrupt contexts on a UP machine running a preemptible kernel.

4. Critical section present in process and interrupt contexts on an SMP machine running a preemptible kernel.

The Old Semaphore Interface

The mutex interface, which replaces the older semaphore interface, originated in the `-rt` tree and was merged into the mainline with the 2.6.16 kernel release. The semaphore interface is still around, however. Basic usage of the semaphore interface is as follows:

```
#include <asm/semaphore.h>  /* Architecture dependent
                               header */

/* Statically declare a semaphore. To dynamically
   create a semaphore, use init_MUTEX() */
static DECLARE_MUTEX(mysem);

down(&mysem);    /* Acquire the semaphore */

/* ... Critical Section code ... */

up(&mysem);      /* Release the semaphore */
```

Semaphores can be configured to allow a predetermined number of threads into the critical section simultaneously. However, semaphores that permit more than a single holder at a time are rarely used.

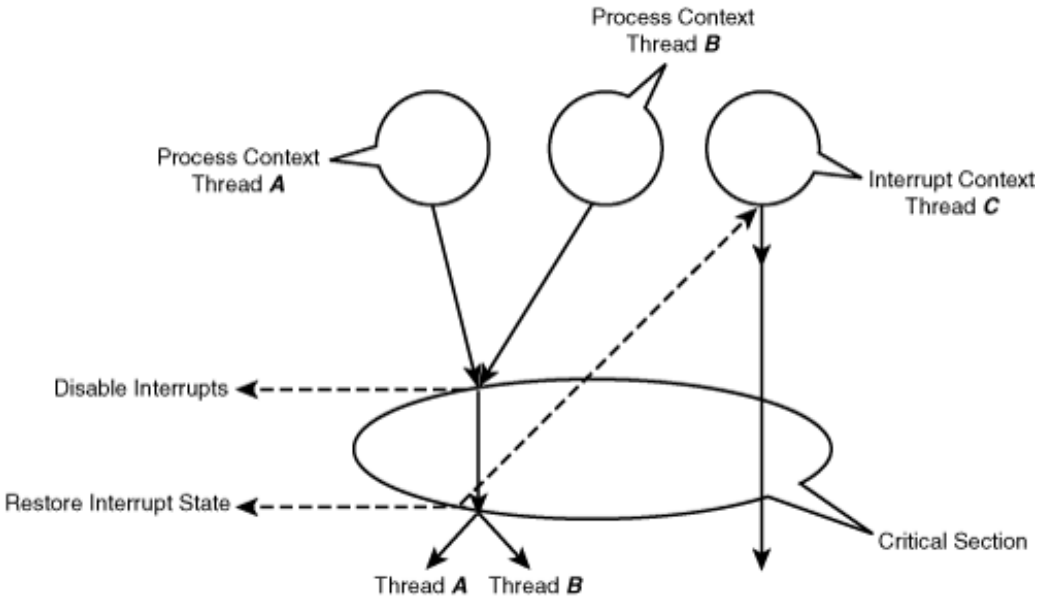
Case 1: Process Context, UP Machine, No Preemption

This is the simplest case and needs no locking, so we won't discuss this further.

Case 2: Process and Interrupt Contexts, UP Machine, No Preemption

In this case, you need to disable only interrupts to protect the critical region. To see why, assume that A and B are process context threads, and C is an interrupt context thread, all vying to enter the same critical section, as shown in Figure 2.4.

Figure 2.4. Process and interrupt context threads inside a critical section.



Because Thread C is executing in interrupt context and always runs to completion before yielding to Thread A or Thread B, it need not worry about protection. Thread A, for its part, need not be concerned about Thread B (and vice versa) because the kernel is not preemptible. Thus, Thread A and Thread B need to guard against only the possibility of Thread C stomping through the critical section while they are inside the same section. They achieve this by disabling interrupts prior to entering the critical section:

```
Point A:
  local_irq_disable();  /* Disable Interrupts in local CPU */
  /* ... Critical Section ...  */
  local_irq_enable();   /* Enable Interrupts in local CPU */
```

However, if interrupts were already disabled when execution reached Point A, `local_irq_enable()` creates the unpleasant side effect of reenabling interrupts, rather than restoring interrupt state. This can be fixed as follows:

```
unsigned long flags;

Point A:
  local_irq_save(flags);     /* Disable Interrupts */
  /* ... Critical Section ... */
  local_irq_restore(flags);  /* Restore state to what
                                it was at Point A */
```

This works correctly irrespective of the interrupt state at Point A.

Case 3: Process and Interrupt Contexts, UP Machine, Preemption

If preemption is enabled, mere disabling of interrupts won't protect your critical region from being trampled over. There is the possibility of multiple threads simultaneously entering the critical section in process context. Referring back to Figure 2.4 in this scenario, Thread A and Thread B now need to protect themselves against each other in addition to guarding against Thread C. The solution apparently, is to disable kernel preemption before the start of the critical section and reenable it at the end, in addition to disabling/reenabling interrupts.

For this, Thread A and Thread B use the *irq* variant of spinlocks:

```
unsigned long flags;

Point A:
  /* Save interrupt state.
   * Disable interrupts - this implicitly disables preemption */
  spin_lock_irqsave(&mylock, flags);

  /* ... Critical Section ... */

  /* Restore interrupt state to what it was at Point A */
  spin_unlock_irqrestore(&mylock, flags);
```

Preemption state need not be explicitly restored to what it was at Point A because the kernel internally does that for you via a variable called the *preemption counter.* The counter gets incremented whenever preemption is disabled (using `preempt_disable()`) and gets decremented whenever preemption is enabled (using `preempt_enable()`). Preemption kicks in only if the counter value is zero.

Case 4: Process and Interrupt Contexts, SMP Machine, Preemption

Let's now assume that the critical section executes on an SMP machine. Your kernel has been configured with `CONFIG_SMP` and `CONFIG_PREEMPT` turned on.

In the scenarios discussed this far, spinlock primitives have done little more than enable/disable preemption and interrupts. The actual locking functionality has been compiled away. In the presence of SMP, the locking logic gets compiled in, and the spinlock primitives are rendered SMP-safe. The SMP-enabled semantics is as follows:

```
unsigned long flags;

Point A:
  /*
    - Save interrupt state on the local CPU
    - Disable interrupts on the local CPU. This implicitly disables
      preemption.
    - Lock the section to regulate access by other CPUs
   */
  spin_lock_irqsave(&mylock, flags);

  /* ... Critical Section ... */

  /*
    - Restore interrupt state and preemption to what it
      was at Point A for the local CPU
    - Release the lock
   */
  spin_unlock_irqrestore(&mylock, flags);
```

On SMP systems, only interrupts on the local CPU are disabled when a spinlock is acquired. So, a process context thread (say Thread A in Figure 2.4) might be running on one CPU, while an interrupt handler (say Thread C in Figure 2.4) is executing on another CPU. An interrupt handler on a nonlocal processor thus needs to spin-wait until the process context code on the local processor exits the critical section. The interrupt context code calls `spin_lock()`/`spin_unlock()` to do this:

```
spin_lock(&mylock);

/* ... Critical Section ... */

spin_unlock(&mylock);
```

Similar to the irq variants, spinlocks also have *bottom half* (BH) flavors. `spin_lock_bh()` disables bottom halves when the lock is acquired, whereas `spin_unlock_bh()` reenables bottom halves when the lock is released. We discuss bottom halves in Chapter 4.

> **The `-rt` tree**
>
> The real time (`-rt`) tree, also called the `CONFIG_PREEMPT_RT` patch-set, implements low-latency modifications to the kernel. The patch-set, downloadable from www.kernel.org/pub/linux/kernel/projects/rt, allows most of the kernel to be preempted, partly by replacing many spinlocks with mutexes. It also incorporates high-resolution timers. Several `-rt` features have been integrated into the mainline kernel. You will find detailed documentation at the project's wiki hosted at http://rt.wiki.kernel.org/.

The kernel has specialized locking primitives in its repertoire that help improve performance under specific conditions. Using a mutual-exclusion scheme tailored to your needs makes your code more powerful. Let's take a look at some of the specialized exclusion mechanisms.

Atomic Operators

Atomic operators are used to perform lightweight one-shot operations such as bumping counters, conditional increments, and setting bit positions. Atomic operations are guaranteed to be serialized and do not need locks for protection against concurrent access. The implementation of atomic operators is architecture-dependent.

To check whether there are any remaining data references before freeing a kernel network buffer (called an `skbuff`), the `skb_release_data()` routine defined in *net/core/skbuff.c* does the following:

```
if (!skb->cloned ||
  /* Atomically decrement and check if the returned value is zero */
    !atomic_sub_return(skb->nohdr ? (1 << SKB_DATAREF_SHIFT) + 1 :
                       1,&skb_shinfo(skb)->dataref)) {
  /* ... */
  kfree(skb->head);
}
```

While `skb_release_data()` is thus executing, another thread using `skbuff_clone()` (defined in the same file) might be simultaneously incrementing the data reference counter:

```
/* ... */
/* Atomically bump up the data reference count */
atomic_inc(&(skb_shinfo(skb)->dataref));
/* ... */
```

The use of atomic operators protects the data reference counter from being trampled by these two threads. It

also eliminates the hassle of using locks to protect a single integer variable from concurrent access.

The kernel also supports operators such as `set_bit()`, `clear_bit()`, and `test_and_set_bit()` to atomically engage in bit manipulations. Look at *include/asm-your-arch/atomic.h* for the atomic operators supported on your architecture.

Reader-Writer Locks

Another specialized concurrency regulation mechanism is a reader-writer variant of spinlocks. If the usage of a critical section is such that separate threads either read from or write to a shared data structure, but don't do both, these locks are a natural fit. Multiple reader threads are allowed inside a critical region simultaneously. Reader spinlocks are defined as follows:

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;

read_lock(&myrwlock);     /* Acquire reader lock */
/* ... Critical Region ... */
read_unlock(&myrwlock);   /* Release lock */
```

However, if a writer thread enters a critical section, other reader or writer threads are not allowed inside. To use writer spinlocks, you would write this:

```
rwlock_t myrwlock = RW_LOCK_UNLOCKED;

write_lock(&myrwlock);    /* Acquire writer lock */
/* ... Critical Region ... */
write_unlock(&myrwlock);  /* Release lock */
```

Look at the IPX routing code present in *net/ipx/ipx_route.c* for a real-life example of a reader-writer spinlock. A reader-writer lock called `ipx_routes_lock` protects the IPX routing table from simultaneous access. Threads that need to look up the routing table to forward packets request reader locks. Threads that need to add or delete entries from the routing table acquire writer locks. This improves performance because there are usually far more instances of routing table lookups than routing table updates.

Like regular spinlocks, reader-writer locks also have corresponding irq variants—namely, `read_lock_irqsave()`, `read_lock_irqrestore()`, `write_lock_irqsave()`, and `write_lock_irqrestore()`. The semantics of these functions are similar to those of regular spinlocks.

Sequence locks or *seqlocks*, introduced in the 2.6 kernel, are reader-writer locks where writers are favored over readers. This is useful if write operations on a variable far outnumber read accesses. An example is the `jiffies_64` variable discussed earlier in this chapter. Writer threads do not wait for readers who may be inside a critical section. Because of this, reader threads may discover that their entry inside a critical section has failed and may need to retry:

```
u64 get_jiffies_64(void) /* Defined in kernel/time.c */
{
  unsigned long seq;
  u64 ret;
  do {
    seq = read_seqbegin(&xtime_lock);
    ret = jiffies_64;
  } while (read_seqretry(&xtime_lock, seq));
  return ret;
}
```

Writers protect critical regions using `write_seqlock()` and `write_sequnlock()`.

The 2.6 kernel introduced another mechanism called *Read-Copy Update* (RCU), which yields improved performance when readers far outnumber writers. The basic idea is that reader threads can execute without locking. Writer threads are more complex. They perform update operations on a copy of the data structure and replace the pointer that readers see. The original copy is maintained until the next context switch on all CPUs to ensure completion of all ongoing read operations. Be aware that using RCU is more involved than using the primitives discussed thus far and should be used only if you are sure that it's the right tool for the job. RCU data structures and interface functions are defined in *include/linux/rcupdate.h.* There is ample documentation in *Documentation/RCU/**.

For an RCU usage example, look at *fs/dcache.c.* On Linux, each file is associated with directory entry information (stored in a structure called *dentry*), metadata information (stored in an inode), and actual data (stored in data blocks). Each time you operate on a file, the components in the file path are parsed, and the corresponding dentries are obtained. The dentries are kept cached in a data structure called the *dcache*, to speed up future operations. At any time, the number of dcache lookups is much more than dcache updates, so references to the dcache are protected using RCU primitives.

Debugging

Concurrency-related problems are generally hard to debug because they are usually difficult to reproduce. It's a good idea to enable SMP (`CONFIG_SMP`) and preemption (`CONFIG_PREEMPT`) while compiling and testing your code, even if your production kernel is going to run on a UP machine with preemption disabled. There is a kernel configuration option under *Kernel hacking* called *Spinlock and rw-lock debugging* (`CONFIG_DEBUG_SPINLOCK`) that can help you catch some common spinlock errors. Also available are tools such as lockmeter (http://oss.sgi.com/projects/lockmeter/) that collect lock-related statistics.

A common concurrency problem occurs when you forget to lock an access to a shared resource. This results in different threads "racing" through that access in an unregulated manner. The problem, called a *race condition*, might manifest in the form of occasional strange code behavior.

Another potential problem arises when you miss releasing held locks in certain code paths, resulting in deadlocks. To understand this, consider the following example:

```
spin_lock(&mylock);     /* Acquire lock */

/* ... Critical Section ... */

if (error) {            /* This error condition occurs rarely */
  return -EIO; /* Forgot to release the lock! */
}

spin_unlock(&mylock);   /* Release lock */
```

After the occurrence of the error condition, any thread trying to acquire `mylock` gets deadlocked, and the kernel might freeze.

If the problem first manifests months or years after you write the code, it'll be all the more tough to go back and debug it. (There is a related debugging example in the section "Kdump" in Chapter 21, "Debugging Device Drivers.") To avoid such unpleasant encounters, concurrency logic should be designed when you architect your software.

Process Filesystem

The process filesystem (*procfs*) is a virtual filesystem that creates windows into the innards of the kernel. The data you see when you browse procfs is generated by the kernel on-the-fly. Files in procfs are used to configure kernel parameters, look at kernel structures, glean statistics from device drivers, and get general system information.

Procfs is a pseudo filesystem. This means that files resident in procfs are not associated with physical storage devices such as hard disks. Instead, data in those files is dynamically created on demand by the corresponding entry points in the kernel. Because of this, file sizes in procfs get shown as zero. Procfs is usually mounted under the */proc* directory during kernel boot; you can see this by invoking the `mount` command.

To get a first feel of the capabilities of procfs, examine the contents of */proc/cpuinfo*, */proc/meminfo*, */proc/interrupts*, */proc/tty/driver/serial*, */proc/bus/usb/devices*, and */proc/stat*. Certain kernel parameters can be changed at runtime by writing to files under */proc/sys/*. For example, you can change kernel `printk` log levels by echoing a new set of values to */proc/sys/kernel/printk*. Many utilities (such as ps) and system performance monitoring tools (such as sysstat) internally derive information from files residing under */proc*.

Seq files, introduced in the 2.6 kernel, simplify large procfs operations. They are described in Appendix C, "Seq Files."

Allocating Memory

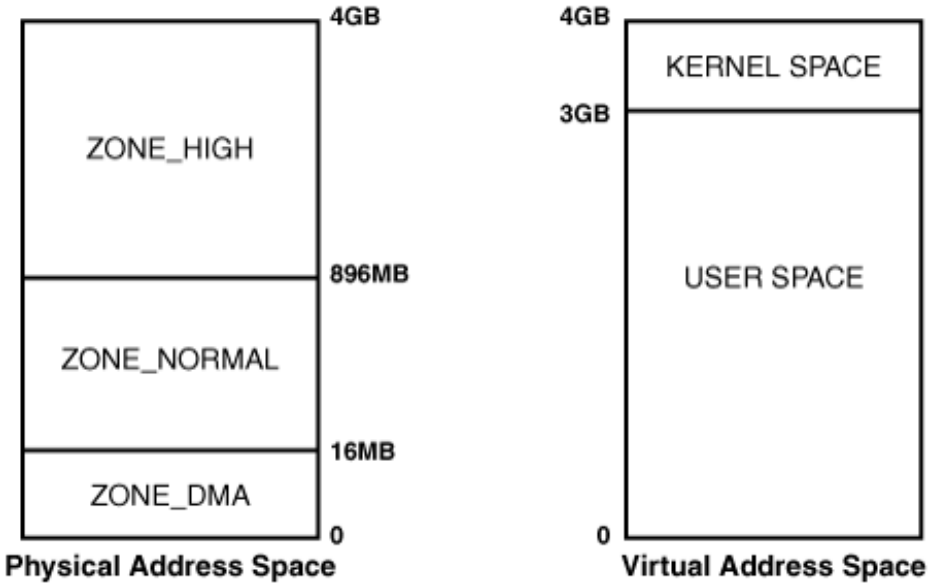
Some device drivers have to be aware of the existence of *memory zones.* In addition, many drivers need the services of memory-allocation functions. In this section, let's briefly discuss both.

The kernel organizes physical memory into *pages.* The page size depends on the architecture. On x86-based machines, it's 4096 bytes. Each page in physical memory has a `struct page` (defined in *include/linux/mm_types.h*) associated with it:

```
struct page {
  unsigned long flags; /* Page status */
  atomic_t _count;     /* Reference count */
  /* ... */
  void * virtual;      /* Explained later on */
};
```

On 32-bit x86 systems, the default kernel configuration splits the available 4GB address space into a 3GB virtual memory space for user processes and a 1GB space for the kernel, as shown in Figure 2.5. This imposes a 1GB limit on the amount of physical memory that the kernel can handle. In reality, the limit is 896MB because 128MB of the address space is occupied by kernel data structures. You may increase this limit by changing the 3GB/1GB split during kernel configuration, but you will incur the displeasure of memory-intensive applications if you reduce the virtual address space of user processes.

Figure 2.5. Default address space split on a 32-bit PC system.



Kernel addresses that map the low 896MB differ from physical addresses by a constant offset and are called *logical addresses.* With "high memory" support, the kernel can access memory beyond 896MB by generating *virtual addresses* for those regions using special mappings. All logical addresses are kernel virtual addresses, but not vice versa.

This leads us to the following kernel memory zones:

1. `ZONE_DMA` (<16MB), the zone used for *Direct Memory Access* (DMA). Because legacy ISA devices have 24 address lines and can access only the first 16MB, the kernel tries to dedicate this area for such devices.

2. `ZONE_NORMAL` (16MB to 896MB), the normally addressable region, also called low memory. The "virtual" field in `struct page` for low memory pages contains the corresponding logical addresses.

3. `ZONE_HIGH` (>896MB), the space that the kernel can access only after mapping resident pages to regions in `ZONE_NORMAL` (using `kmap()` and `kunmap()`). The corresponding kernel addresses are virtual and not logical. The "virtual" field in `struct page` for high memory pages points to `NULL` if the page is not kmapped.

`kmalloc()` is a memory-allocation function that returns contiguous memory from `ZONE_NORMAL`. The prototype is as follows:

```
void *kmalloc(int count, int flags);
```

Where `count` is the number of bytes to allocate, and `flags` is a mode specifier. All supported flags are listed in *include/linux./gfp.h* (gfp stands for *get free pages*), but these are the commonly used ones:

1. `GFP_KERNEL` Used by process context code to allocate memory. If this flag is specified, `kmalloc()` is allowed to go to sleep and wait for pages to get freed up.

2. `GFP_ATOMIC` Used by interrupt context code to get hold of memory. In this mode, `kmalloc()` is not allowed to sleep-wait for free pages, so the probability of successful allocation with `GFP_ATOMIC` is lower than with `GFP_KERNEL`.

Because memory returned by `kmalloc()` retains the contents from its previous incarnation, there could be a security risk if it's exposed to user space. To get zeroed kmalloced memory, use `kzalloc()`.

If you need to allocate large memory buffers, and you don't require the memory to be physically contiguous, use `vmalloc()` rather than `kmalloc()`:

```
void *vmalloc(unsigned long count);
```

Here `count` is the requested allocation size. The function returns kernel virtual addresses.

`vmalloc()` enjoys bigger allocation size limits than `kmalloc()` but is slower and can't be called from interrupt context. Moreover, you cannot use the physically discontiguous memory returned by `vmalloc()` to perform *Direct Memory Access* (DMA). High-performance network drivers commonly use `vmalloc()` to allocate large descriptor rings when the device is opened.

The kernel offers more sophisticated memory allocation techniques. These include *look aside buffers*, *slabs*, and *mempools*, which are beyond the scope of this chapter.

Looking at the Sources

Kernel boot starts with the execution of real mode assembly code living in the *arch/x86/boot/* directory. Look at *arch/x86/kernel/setup_32.c* to see how the protected mode kernel obtains information gleaned by the real mode kernel.

The first boot message is printed by code residing in *init/main.c.* Dig inside *init/calibrate.c* to learn more about BogoMIPS calibration and *include/asm-your-arch/bugs.h* for an insight into architecture-dependent checks.

Timer services in the kernel consist of architecture-dependent portions that live in *arch/your-arch/kernel/* and generic portions implemented in *kernel/timer.c.* For related definitions, look at the header files, *include/linux/time*.h*.

`jiffies` is defined in *linux/jiffies.h.* The value for `HZ` is processor-dependent and can be found in *include/asm-your-arch/param.h*.

Memory management sources reside in the top-level *mm/* directory.

Table 2.1 contains a summary of the main data structures used in this chapter and the location of their definitions in the source tree. Table 2.2 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 2.1. Summary of Data Structures

Data Structure	Location	Description
`HZ`	*include/asm-your-arch/param.h*	Number of times the system timer ticks in 1 second
`loops_per_jiffy`	*init/main.c*	Number of times the processor executes an internal delay-loop in 1 jiffy
`timer_list`	*include/linux/timer.h*	Used to hold the address of a routine that you want to execute at some point in the future
`timeval`	*include/linux/time.h*	Timestamp
`spinlock_t`	*include/linux/spinlock_types.h*	A busy-locking mechanism to ensure that only a single thread enters a critical section
`semaphore`	*include/asm-your-arch/semaphore.h*	A sleep-locking mechanism that allows a predetermined number of users to enter a critical section
`mutex`	*include/linux/mutex.h*	The new interface that replaces `semaphore`
`rwlock_t`	*include/linux/spinlock_types.h*	Reader-writer spinlock
`page`	*include/linux/mm_types.h*	Kernel's representation of a physical memory page

Table 2.2. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description

`time_after()`
`time_after_eq()`
`time_before()`
`ime_before_eq()`
	include/linux/jiffies.h	Compares the current value of `jiffies` with a specified future value

`schedule_timeout()`	*kernel/timer.c*	Schedules a process to run after a specified timeout has elapsed
`wait_event_timeout()`	*include/linux/wait.h*	Resumes execution if a specified condition becomes true or if a timeout occurs
`DEFINE_TIMER()`	*include/linux/timer.h*	Statically defines a timer
`init_timer()`	*kernel/timer.c*	Dynamically defines a timer
`add_timer()`	*include/linux/timer.h*	Schedules the timer for execution after the timeout has elapsed
`mod_timer()`	*kernel/timer.c*	Changes timer expiration
`timer_pending()`	*include/linux/timer.h*	Checks if a timer is pending at the moment
`udelay()`	*include/asm-your-arch/delay.h arch/your-arch/lib/delay.c*	Busy-waits for the specified number of microseconds
`rdtsc()`	*include/asm-x86/msr.h*	Gets the value of the TSC on Pentium-compatible processors
`do_gettimeofday()`	*kernel/time.c*	Obtains wall time
`local_irq_disable()`	*include/asm-your-arch/system.h*	Disables interrupts on the local CPU
`local_irq_enable()`	*include/asm-your-arch/system.h*	Enables interrupts on the local CPU
`local_irq_save()`	*include/asm-your-arch/system.h*	Saves interrupt state and disables interrupts
`local_irq_restore()`	*include/asm-your-arch/system.h*	Restores interrupt state to what it was when the matching `local_irq_save()` was called
`spin_lock()`	*include/linux/spinlock.h kernel/spinlock.c*	Acquires a spinlock.
`spin_unlock()`	*include/linux/spinlock.h*	Releases a spinlock
`spin_lock_irqsave()`	*include/linux/spinlock.h kernel/spinlock.c*	Saves interrupt state, disables interrupts and preemption on local CPU, and locks their critical section to regulate access by other CPUs
`spin_unlock_irqrestore()`	*include/linux/spinlock.h kernel/spinlock.c*	Restores interrupt state and preemption and releases the lock
`DEFINE_MUTEX()`	*include/linux/mutex.h*	Statically declares a mutex
`mutex_init()`	*include/linux/mutex.h*	Dynamically declares a mutex
`mutex_lock()`	*kernel/mutex.c*	Acquires a mutex

Kernel Interface	Location	Description
`mutex_unlock()`	*kernel/mutex.c*	Releases a mutex
`DECLARE_MUTEX()`	*include/asm-your-arch/semaphore.h*	Statically declares a semaphore
`init_MUTEX()`	*include/asm-your-arch/semaphore.h*	Dynamically declares a semaphore
`up()`	*arch/your-arch/kernel/semaphore.c*	Acquires a semaphore
`down()`	*arch/your-arch/kernel/semaphore.c*	Releases a semaphore

`atomic_inc()`
`atomic_inc_and_test()`
`atomic_dec()`
`atomic_dec_and_test()`
`clear_bit()`
`set_bit()`
`test_bit()`
`test_and_set_bit()`
	include/asm-your-arch/atomic.h	Atomic operators to perform lightweight operations

`read_lock()`
`read_unlock()`
`read_lock_irqsave()`
`read_lock_irqrestore()`
`write_lock()`
`write_unlock()`
`write_lock_irqsave()`
`write_lock_irqrestore()`
	include/linux/spinlock.h
	kernel/spinlock.c
		Reader-writer variant of spinlocks

`down_read()`
`up_read()`
`down_write()`
`up_write()`
	kernel/rwsem.c	Reader-writer variant of semaphores

`read_seqbegin()`
`read_seqretry()`
`write_seqlock()`
`write_sequnlock()`
	include/linux/seqlock.h	Seqlock operations

`kmalloc()`	*include/linux/slab.h* *mm/slab.c*	Allocates physically contiguous memory from `ZONE_NORMAL`
`kzalloc()`	*include/linux/slab.h* *mm/util.c*	Obtains zeroed kmalloced memory
`kfree()`	*mm/slab.c*	Releases kmalloced memory
`vmalloc()`	*mm/vmalloc.c*	Allocates virtually contiguous memory that is not guaranteed to be physically contiguous.

Chapter 3. Kernel Facilities

In This Chapter

- Kernel Threads 56
- Helper Interfaces 65
- Looking at the Sources 85

In this chapter, let's look at some kernel facilities that are useful components in a driver developer's toolbox. We start this chapter by looking at a kernel facility that is similar to user processes; kernel threads are programming abstractions oriented toward concurrent processing.

The kernel offers several helper interfaces that simplify your code, eliminate redundancies, increase code readability, and help in long-term maintenance. We will look at linked lists, hash lists, work queues, notifier chains, completion functions, and error-handling aids. These helpers are bug free and optimized, so your driver also inherits those benefits for free.

Kernel Threads

A *kernel thread* is a way to implement background tasks inside the kernel. The task can be busy handling asynchronous events or sleep-waiting for an event to occur. Kernel threads are similar to user processes, except that they live in kernel space and have access to kernel functions and data structures. Like user processes, kernel threads have the illusion of monopolizing the processor because of preemptive scheduling. Many device drivers utilize the services of kernel threads to implement assistant or helper tasks. For example, the *khubd* kernel thread, which is part of the Linux USB driver core (covered in Chapter 11, "Universal Serial Bus") monitors USB hubs and configures USB devices when they are hot-plugged into the system.

Creating a Kernel Thread

Let's learn about kernel threads with the help of an example. While developing the example thread, you will also learn about kernel concepts such as process states, wait queues, and user mode helpers. When you are comfortable with kernel threads, you can use them as a test vehicle for carrying out various experiments within the kernel.

Assume that you would like the kernel to asynchronously invoke a user mode program to send you an email or pager alert, whenever it senses that the health of certain key kernel data structures is deteriorating. (For instance, free space in network receive buffers has dipped below a low watermark.)

This is a candidate for being implemented as a kernel thread for the following reasons:

- It's a background task because it has to wait for asynchronous events.
- It needs access to kernel data structures because the actual detection of events is done by other parts of the kernel.
- It has to invoke a user mode helper program, which is a time-consuming operation.

Built-In Kernel Threads

To see the kernel threads (also called *kernel processes*) running on your system, run the `ps` command. Names of kernel threads are surrounded by square brackets:

```
bash> ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 22:36 ?       00:00:00 init [3]
root         2     0  0 22:36 ?       00:00:00 [kthreadd]
root         3     2  0 22:36 ?       00:00:00 [ksoftirqd/0]
root         4     2  0 22:36 ?       00:00:00 [events/0]
root        38     2  0 22:36 ?       00:00:00 [pdflush]
root        39     2  0 22:36 ?       00:00:00 [pdflush]
root        29     2  0 22:36 ?       00:00:00 [khubd]
root       695     2  0 22:36 ?       00:00:00 [kjournald]
...
root      3914     2  0 22:37 ?       00:00:00 [nfsd]
root      3915     2  0 22:37 ?       00:00:00 [nfsd]
...
root      4015  3364  0 22:55 tty3    00:00:00 -bash
root      4066  4015  0 22:59 tty3    00:00:00 ps -ef
```

The *[ksoftirqd/0]* kernel thread is an aid to implement *softirqs.* Softirqs are raised by interrupt handlers to request "bottom half" processing of portions of the handler whose execution can be deferred. We take a detailed look at bottom halves and softirqs in Chapter 4, "Laying the Groundwork," but the basic idea here is to allow as little code as possible to be present inside interrupt handlers. Having small interrupt handlers reduces interrupt-off times in the system, resulting in lower latencies. *Ksoftirqd*'s job is to ensure that a high load of softirqs neither starves the softirqs nor overwhelms the system. On *Symmetric Multi Processing* (SMP) machines where multiple thread instances can run on different processors in parallel, one instance of ksoftirqd is created per CPU to improve throughput (*ksoftirqd/n*, where *n* is the CPU number).

The *events/n* threads (where *n* is the CPU number) help implement *work queues*, which are another way of deferring work in the kernel. Parts of the kernel that desire deferred execution of work can either create their own work queue or make use of the default *events/n* worker thread. Work queues are also dissected in Chapter 4.

The task of the *pdflush* kernel thread is to flush out dirty pages from the page cache. The page cache buffers accesses to the disk. To improve performance, actual writes to the disk are delayed until the pdflush daemon writes out dirtied data to disk. This is done if the available free memory dips below a threshold, or if the page has remained dirty for a sufficiently long time. In 2.4 kernels, these two tasks were respectively performed by separate kernel threads, *bdflush* and

kupdated. You might have noticed two instances of pdflush in the ps output. A new instance is created if the kernel senses that existing instances have their hands full, servicing disk queues. This improves throughput, especially if your system has multiple disks and many of them are busy.

As you saw in the preceding chapter, kjournald is the generic kernel journaling thread, which is used by filesystems such as EXT3.

The Linux *Network File System* (NFS) server is implemented using a set of kernel threads named *nfsd*.

Our example kernel thread relinquishes the processor until it gets woken up by parts of the kernel responsible for monitoring the data structures of interest. When awake, it invokes a user mode helper program and passes appropriate identity codes in its environment.

To create a kernel thread, use `kernel_thread()`:

```
ret = kernel_thread(mykthread, NULL,
            CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
```

The flags specify the resources to be shared between the parent and child threads. `CLONE_FILES` specifies that open files are to be shared, and `CLONE_SIGHAND` requests that signal handlers be shared.

Listing 3.1 shows the example implementation. Because kernel threads usually act as helpers to device drivers, they are created when the driver is initialized. In this case, however, the example thread can be created from any suitable place, for instance, *init/main.c*.

The thread starts by invoking `daemonize()`, which performs initial housekeeping and changes the parent of the calling thread to a kernel thread called *kthreadd.* Each Linux thread has a single parent. If a parent process dies without waiting for its child to exit, the child becomes a *zombie* process and wastes resources. Reparenting the child to kthreadd, avoids this and ensures proper cleanup when the thread exits.[1]

[1] In 2.6.21 and earlier kernels, `daemonize()` reparents the calling thread to the init task by calling `reparent_to_init()`.

Because `daemonize()` blocks all signals by default, use `allow_signal()` to enable delivery if your thread desires to handle a particular signal. There are no signal handlers inside the kernel, so use `signal_pending()` to check for signals and take appropriate action. For debugging purposes, the code in Listing 3.1 requests delivery of `SIGKILL` and dies if it's received.

`kernel_thread()` is depreciated in favor of the higher-level *kthread* API, which is built over the former. We will look at kthreads later on.

Listing 3.1. Implementing a Kernel Thread

Code View:

```
static DECLARE_WAIT_QUEUE_HEAD(myevent_waitqueue);
rwlock_t myevent_lock;
extern unsigned int myevent_id;  /* Holds the identity of the
                                    troubled data structure.
                                    Populated later on */
static int mykthread(void *unused)
{
  unsigned int event_id = 0;
  DECLARE_WAITQUEUE(wait, current);
  /* Become a kernel thread without attached user resources */
  daemonize("mykthread");

  /* Request delivery of SIGKILL */
  allow_signal(SIGKILL);

  /* The thread sleeps on this wait queue until it's
     woken up by parts of the kernel in charge of sensing
     the health of data structures of interest */
  add_wait_queue(&myevent_waitqueue, &wait);

  for (;;) {
    /* Relinquish the processor until the event occurs */
    set_current_state(TASK_INTERRUPTIBLE);
    schedule();  /* Allow other parts of the kernel to run */
    /* Die if I receive SIGKILL */
    if (signal_pending(current)) break;
    /* Control gets here when the thread is woken up */
    read_lock(&myevent_lock);      /* Critical section starts */
    if (myevent_id) { /* Guard against spurious wakeups */
      event_id = myevent_id;
      read_unlock(&myevent_lock); /* Critical section ends */
      /* Invoke the registered user mode helper and
         pass the identity code in its environment */
      run_umode_handler(event_id); /* Expanded later on */
    } else {
      read_unlock(&myevent_lock);
    }
  }

  set_current_state(TASK_RUNNING);
  remove_wait_queue(&myevent_waitqueue, &wait);
  return 0;
}
```

If you compile and run this as part of the kernel, you can see the newly created thread, *mykthread*, in the ps output:

```
bash> ps -ef
    UID        PID  PPID C STIME TTY       TIME CMD
    root         1     0 0 21:56 ?     00:00:00 init [3]
    root         2     1 0 22:36 ?     00:00:00 [ksoftirqd/0]
    ...
    root         111   1 0 21:56 ?     00:00:00 [mykthread]
    ...
```

Before we delve further into the thread implementation, let's write a code snippet that monitors the health of a data structure of interest and awakens mykthread if a problem condition is detected:

```
/* Executed by parts of the kernel that own the
   data structures whose health you want to monitor */
/* ... */

if (my_key_datastructure looks troubled) {
  write_lock(&myevent_lock);  /* Serialize */
  /* Fill in the identity of the data structure */
  myevent_id = datastructure_id;

  write_unlock(&myevent_lock);

  /* Wake up mykthread */
  wake_up_interruptible(&myevent_waitqueue);
}

/* ... */
```

Listing 3.1 executes in process context, whereas the preceding snippet runs from either process or interrupt context. Process and interrupt contexts communicate via kernel data structures. Our example uses `myevent_id` and `myevent_waitqueue` for this communication. `myevent_id` contains the identity of the data structure in trouble. Access to `myevent_id` is serialized using locks.

Note that kernel threads are preemptible only if `CONFIG_PREEMPT` is turned on at compile time. If `CONFIG_PREEMPT` is off, or if you are still running a 2.4 kernel without the preemption patch, your thread will freeze the system if it does not go to sleep. If you comment out `schedule()` in Listing 3.1 and disable `CONFIG_PREEMPT` in your kernel configuration, your system will lock up.

You will learn how to obtain soft real-time responses from kernel threads when we discuss scheduling policies in Chapter 19, "Drivers in User Space."

Process States and Wait Queues

Here's the code region from Listing 3.1 that puts mykthread to sleep while waiting for events:

```
add_wait_queue(&myevent_waitqueue, &wait);
for (;;) {

  /* ... */
  set_current_state(TASK_INTERRUPTIBLE);
  schedule();    /* Relinquish the processor */
  /* Point A */

  /* ... */
}
set_current_state(TASK_RUNNING);
remove_wait_queue(&myevent_waitqueue, &wait);
```

The operation of the preceding snippet is based on two concepts: *wait queues* and *process states*.

Wait queues hold threads that need to wait for an event or a system resource. Threads in a wait queue go to sleep until they are woken up by another thread or an interrupt handler that is responsible for detecting the event. Queuing and dequeuing are respectively done using `add_wait_queue()` and `remove_wait_queue()`, and waking up queued tasks is accomplished via `wake_up_interruptible()`.

A kernel thread (or a normal process) can be in any of the following process states: *running*, *interruptible*, *uninterruptible*, *zombie*, *stopped*, *traced*, or *dead*. These states are defined in *include/linux/sched.h*:

- A process in the `running` state (`TASK_RUNNING`) is in the scheduler run queue and is a candidate for getting CPU time allotted by the scheduler.
- A task in the `interruptible` state (`TASK_INTERRUPTIBLE`) is waiting for an event to occur and is not in the scheduler run queue. When the task gets woken up, or if a signal is delivered to it, it re-enters the run queue.
- The `uninterruptible` state (`TASK_UNINTERRUPTIBLE`) is similar to the `interruptible` state except that receipt of a signal will not put the task back into the run queue.
- A `stopped` task (`TASK_STOPPED`) has stopped execution due to receipt of certain signals.
- If an application such as strace is using the ptrace support in the kernel to intercept a task, it'll be in the `traced` state (`TASK_TRACED`).
- A task in the `zombie` state (`EXIT_ZOMBIE`) has terminated, but its parent did not wait for the task to complete. An exiting task is either in the `EXIT_ZOMBIE` state or the dead (`EXIT_DEAD`) state.

You can use `set_current_state()` to set the run state of your kernel thread.

Let's now turn back to the preceding code snippet. mykthread sleeps on a wait queue (`myevent_waitqueue`) and changes its state to `TASK_INTERRUPTIBLE`, signaling its desire to opt out of the scheduler run queue. The call to `schedule()` asks the scheduler to choose and run a new task from its run queue. When code responsible for health monitoring wakes up mykthread using `wake_up_interruptible(&myevent_waitqueue)`, the thread is put back into the scheduler run queue. The process state also gets simultaneously changed to `TASK_RUNNING`, so there is no race condition even if the wake up occurs between the time the task state is set to `TASK_INTERRUPTIBLE` and the time `schedule()` is called. The thread also gets back into the run queue if a `SIGKILL` signal is delivered to it. When the scheduler subsequently picks mykthread from the run queue, execution resumes from Point A.

User Mode Helpers

Mykthread invokes `run_umode_handler()` in Listing 3.1 to notify user space about detected events:

Code View:
```
/* Called from Listing 3.1 */
static void
run_umode_handler(int event_id)
{
  int i = 0;
  char *argv[2], *envp[4], *buffer = NULL;
```

```
  int value;

  argv[i++] = myevent_handler; /* Defined in
                                  kernel/sysctl.c */

  /* Fill in the id corresponding to the data structure
     in trouble */
  if (!(buffer = kmalloc(32, GFP_KERNEL))) return;
  sprintf(buffer, "TROUBLED_DS=%d", event_id);

  /* If no user mode handlers are found, return */
  if (!argv[0]) return; argv[i] = 0;

  /* Prepare the environment for /path/to/helper */
  i = 0;
  envp[i++] = "HOME=/";
  envp[i++] = "PATH=/sbin:/usr/sbin:/bin:/usr/bin";
  envp[i++] = buffer; envp[i]   = 0;

  /* Execute the user mode program, /path/to/helper */
  value = call_usermodehelper(argv[0], argv, envp, 0);

  /* Check return values */
  kfree(buffer);
}
```

The kernel supports a mechanism for requesting user mode programs to help perform certain functions. `run_umode_handler()` uses this facility by invoking `call_usermodehelper()`.

You have to register the user mode program invoked by `run_umode_handler()` via a node in the */proc/sys/* directory. To do so, make sure that `CONFIG_SYSCTL` (files that are part of the */proc/sys/* directory are collectively known as the *sysctl* interface) is enabled in your kernel configuration and add an entry to the `kern_table` array in *kernel/sysctl.c*:

```
{
  .ctl_name     = KERN_MYEVENT_HANDLER, /* Define in
                                           include/linux/sysctl.h */
  .procname     = "myevent_handler",
  .data         = &myevent_handler,
  .maxlen       = 256,
  .mode         = 0644,
  .proc_handler = &proc_dostring,
  .strategy     = &sysctl_string,
},
```

This creates the node */proc/sys/kernel/myevent_handler* in the process filesystem. To register your user mode helper, do the following:

```
bash> echo /path/to/helper > /proc/sys/kernel/myevent_handler
```

This results in */path/to/helper* getting executed when mykthread invokes `run_umode_handler()`.

Mykthread passes the identity of the troubled kernel data structure to the user mode helper through the environment variable `TROUBLED_DS`. The helper can be a simple script like the following that sends you an email alert containing the information it gleaned from its environment:

```
bash> cat /path/to/helper
#!/bin/bash
echo Kernel datastructure $TROUBLED_DS is in trouble | mail -s Alert root
```

`call_usermodehelper()` has to be executed from process context and runs with root privileges. It's implemented using a work queue, which we will soon discuss.

Chapter 3. Kernel Facilities

In This Chapter

- Kernel Threads 56
- Helper Interfaces 65
- Looking at the Sources 85

In this chapter, let's look at some kernel facilities that are useful components in a driver developer's toolbox. We start this chapter by looking at a kernel facility that is similar to user processes; kernel threads are programming abstractions oriented toward concurrent processing.

The kernel offers several helper interfaces that simplify your code, eliminate redundancies, increase code readability, and help in long-term maintenance. We will look at linked lists, hash lists, work queues, notifier chains, completion functions, and error-handling aids. These helpers are bug free and optimized, so your driver also inherits those benefits for free.

Kernel Threads

A *kernel thread* is a way to implement background tasks inside the kernel. The task can be busy handling asynchronous events or sleep-waiting for an event to occur. Kernel threads are similar to user processes, except that they live in kernel space and have access to kernel functions and data structures. Like user processes, kernel threads have the illusion of monopolizing the processor because of preemptive scheduling. Many device drivers utilize the services of kernel threads to implement assistant or helper tasks. For example, the *khubd* kernel thread, which is part of the Linux USB driver core (covered in Chapter 11, "Universal Serial Bus") monitors USB hubs and configures USB devices when they are hot-plugged into the system.

Creating a Kernel Thread

Let's learn about kernel threads with the help of an example. While developing the example thread, you will also learn about kernel concepts such as process states, wait queues, and user mode helpers. When you are comfortable with kernel threads, you can use them as a test vehicle for carrying out various experiments within the kernel.

Assume that you would like the kernel to asynchronously invoke a user mode program to send you an email or pager alert, whenever it senses that the health of certain key kernel data structures is deteriorating. (For instance, free space in network receive buffers has dipped below a low watermark.)

This is a candidate for being implemented as a kernel thread for the following reasons:

- It's a background task because it has to wait for asynchronous events.
- It needs access to kernel data structures because the actual detection of events is done by other parts of the kernel.
- It has to invoke a user mode helper program, which is a time-consuming operation.

Built-In Kernel Threads

To see the kernel threads (also called *kernel processes*) running on your system, run the `ps` command. Names of kernel threads are surrounded by square brackets:

```
bash> ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 22:36 ?       00:00:00 init [3]
root         2     0  0 22:36 ?       00:00:00 [kthreadd]
root         3     2  0 22:36 ?       00:00:00 [ksoftirqd/0]
root         4     2  0 22:36 ?       00:00:00 [events/0]
root        38     2  0 22:36 ?       00:00:00 [pdflush]
root        39     2  0 22:36 ?       00:00:00 [pdflush]
root        29     2  0 22:36 ?       00:00:00 [khubd]
root       695     2  0 22:36 ?       00:00:00 [kjournald]
...
root      3914     2  0 22:37 ?       00:00:00 [nfsd]
root      3915     2  0 22:37 ?       00:00:00 [nfsd]
...
root      4015  3364  0 22:55 tty3    00:00:00 -bash
root      4066  4015  0 22:59 tty3    00:00:00 ps -ef
```

The *[ksoftirqd/0]* kernel thread is an aid to implement *softirqs.* Softirqs are raised by interrupt handlers to request "bottom half" processing of portions of the handler whose execution can be deferred. We take a detailed look at bottom halves and softirqs in Chapter 4, "Laying the Groundwork," but the basic idea here is to allow as little code as possible to be present inside interrupt handlers. Having small interrupt handlers reduces interrupt-off times in the system, resulting in lower latencies. *Ksoftirqd*'s job is to ensure that a high load of softirqs neither starves the softirqs nor overwhelms the system. On *Symmetric Multi Processing* (SMP) machines where multiple thread instances can run on different processors in parallel, one instance of ksoftirqd is created per CPU to improve throughput (*ksoftirqd/n*, where *n* is the CPU number).

The *events/n* threads (where *n* is the CPU number) help implement *work queues*, which are another way of deferring work in the kernel. Parts of the kernel that desire deferred execution of work can either create their own work queue or make use of the default *events/n* worker thread. Work queues are also dissected in Chapter 4.

The task of the *pdflush* kernel thread is to flush out dirty pages from the page cache. The page cache buffers accesses to the disk. To improve performance, actual writes to the disk are delayed until the pdflush daemon writes out dirtied data to disk. This is done if the available free memory dips below a threshold, or if the page has remained dirty for a sufficiently long time. In 2.4 kernels, these two tasks were respectively performed by separate kernel threads, *bdflush* and

kupdated. You might have noticed two instances of pdflush in the ps output. A new instance is created if the kernel senses that existing instances have their hands full, servicing disk queues. This improves throughput, especially if your system has multiple disks and many of them are busy.

As you saw in the preceding chapter, kjournald is the generic kernel journaling thread, which is used by filesystems such as EXT3.

The Linux *Network File System* (NFS) server is implemented using a set of kernel threads named *nfsd*.

Our example kernel thread relinquishes the processor until it gets woken up by parts of the kernel responsible for monitoring the data structures of interest. When awake, it invokes a user mode helper program and passes appropriate identity codes in its environment.

To create a kernel thread, use `kernel_thread()`:

```
ret = kernel_thread(mykthread, NULL,
            CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
```

The flags specify the resources to be shared between the parent and child threads. `CLONE_FILES` specifies that open files are to be shared, and `CLONE_SIGHAND` requests that signal handlers be shared.

Listing 3.1 shows the example implementation. Because kernel threads usually act as helpers to device drivers, they are created when the driver is initialized. In this case, however, the example thread can be created from any suitable place, for instance, *init/main.c*.

The thread starts by invoking `daemonize()`, which performs initial housekeeping and changes the parent of the calling thread to a kernel thread called *kthreadd.* Each Linux thread has a single parent. If a parent process dies without waiting for its child to exit, the child becomes a *zombie* process and wastes resources. Reparenting the child to kthreadd, avoids this and ensures proper cleanup when the thread exits.[1]

[1] In 2.6.21 and earlier kernels, `daemonize()` reparents the calling thread to the init task by calling `reparent_to_init()`.

Because `daemonize()` blocks all signals by default, use `allow_signal()` to enable delivery if your thread desires to handle a particular signal. There are no signal handlers inside the kernel, so use `signal_pending()` to check for signals and take appropriate action. For debugging purposes, the code in Listing 3.1 requests delivery of `SIGKILL` and dies if it's received.

`kernel_thread()` is depreciated in favor of the higher-level *kthread* API, which is built over the former. We will look at kthreads later on.

Listing 3.1. Implementing a Kernel Thread

Code View:

```
static DECLARE_WAIT_QUEUE_HEAD(myevent_waitqueue);
rwlock_t myevent_lock;
extern unsigned int myevent_id;  /* Holds the identity of the
                                    troubled data structure.
                                    Populated later on */
static int mykthread(void *unused)
{
  unsigned int event_id = 0;
  DECLARE_WAITQUEUE(wait, current);
  /* Become a kernel thread without attached user resources */
  daemonize("mykthread");

  /* Request delivery of SIGKILL */
  allow_signal(SIGKILL);

  /* The thread sleeps on this wait queue until it's
     woken up by parts of the kernel in charge of sensing
     the health of data structures of interest */
  add_wait_queue(&myevent_waitqueue, &wait);

  for (;;) {
    /* Relinquish the processor until the event occurs */
    set_current_state(TASK_INTERRUPTIBLE);
    schedule();  /* Allow other parts of the kernel to run */
    /* Die if I receive SIGKILL */
    if (signal_pending(current)) break;
    /* Control gets here when the thread is woken up */
    read_lock(&myevent_lock);      /* Critical section starts */
    if (myevent_id) { /* Guard against spurious wakeups */
      event_id = myevent_id;
      read_unlock(&myevent_lock); /* Critical section ends */
      /* Invoke the registered user mode helper and
         pass the identity code in its environment */
      run_umode_handler(event_id); /* Expanded later on */
    } else {
      read_unlock(&myevent_lock);
    }
  }

  set_current_state(TASK_RUNNING);
  remove_wait_queue(&myevent_waitqueue, &wait);
  return 0;
}
```

If you compile and run this as part of the kernel, you can see the newly created thread, *mykthread*, in the ps output:

```
bash> ps -ef
    UID        PID  PPID C STIME TTY       TIME CMD
    root         1     0 0 21:56 ?     00:00:00 init [3]
    root         2     1 0 22:36 ?     00:00:00 [ksoftirqd/0]
    ...
    root         111   1 0 21:56 ?     00:00:00 [mykthread]
    ...
```

Before we delve further into the thread implementation, let's write a code snippet that monitors the health of a data structure of interest and awakens mykthread if a problem condition is detected:

```
/* Executed by parts of the kernel that own the
   data structures whose health you want to monitor */
/* ... */

if (my_key_datastructure looks troubled) {
  write_lock(&myevent_lock);  /* Serialize */
  /* Fill in the identity of the data structure */
  myevent_id = datastructure_id;

  write_unlock(&myevent_lock);

  /* Wake up mykthread */
  wake_up_interruptible(&myevent_waitqueue);
}

/* ... */
```

Listing 3.1 executes in process context, whereas the preceding snippet runs from either process or interrupt context. Process and interrupt contexts communicate via kernel data structures. Our example uses `myevent_id` and `myevent_waitqueue` for this communication. `myevent_id` contains the identity of the data structure in trouble. Access to `myevent_id` is serialized using locks.

Note that kernel threads are preemptible only if `CONFIG_PREEMPT` is turned on at compile time. If `CONFIG_PREEMPT` is off, or if you are still running a 2.4 kernel without the preemption patch, your thread will freeze the system if it does not go to sleep. If you comment out `schedule()` in Listing 3.1 and disable `CONFIG_PREEMPT` in your kernel configuration, your system will lock up.

You will learn how to obtain soft real-time responses from kernel threads when we discuss scheduling policies in Chapter 19, "Drivers in User Space."

Process States and Wait Queues

Here's the code region from Listing 3.1 that puts mykthread to sleep while waiting for events:

```
add_wait_queue(&myevent_waitqueue, &wait);
for (;;) {

  /* ... */
  set_current_state(TASK_INTERRUPTIBLE);
  schedule();    /* Relinquish the processor */
  /* Point A */

  /* ... */
}
set_current_state(TASK_RUNNING);
remove_wait_queue(&myevent_waitqueue, &wait);
```

The operation of the preceding snippet is based on two concepts: *wait queues* and *process states*.

Wait queues hold threads that need to wait for an event or a system resource. Threads in a wait queue go to sleep until they are woken up by another thread or an interrupt handler that is responsible for detecting the event. Queuing and dequeuing are respectively done using `add_wait_queue()` and `remove_wait_queue()`, and waking up queued tasks is accomplished via `wake_up_interruptible()`.

A kernel thread (or a normal process) can be in any of the following process states: *running*, *interruptible*, *uninterruptible*, *zombie*, *stopped*, *traced*, or *dead*. These states are defined in *include/linux/sched.h*:

- A process in the `running` state (`TASK_RUNNING`) is in the scheduler run queue and is a candidate for getting CPU time allotted by the scheduler.
- A task in the `interruptible` state (`TASK_INTERRUPTIBLE`) is waiting for an event to occur and is not in the scheduler run queue. When the task gets woken up, or if a signal is delivered to it, it re-enters the run queue.
- The `uninterruptible` state (`TASK_UNINTERRUPTIBLE`) is similar to the `interruptible` state except that receipt of a signal will not put the task back into the run queue.
- A `stopped` task (`TASK_STOPPED`) has stopped execution due to receipt of certain signals.
- If an application such as strace is using the ptrace support in the kernel to intercept a task, it'll be in the `traced` state (`TASK_TRACED`).
- A task in the `zombie` state (`EXIT_ZOMBIE`) has terminated, but its parent did not wait for the task to complete. An exiting task is either in the `EXIT_ZOMBIE` state or the dead (`EXIT_DEAD`) state.

You can use `set_current_state()` to set the run state of your kernel thread.

Let's now turn back to the preceding code snippet. mykthread sleeps on a wait queue (`myevent_waitqueue`) and changes its state to `TASK_INTERRUPTIBLE`, signaling its desire to opt out of the scheduler run queue. The call to `schedule()` asks the scheduler to choose and run a new task from its run queue. When code responsible for health monitoring wakes up mykthread using `wake_up_interruptible(&myevent_waitqueue)`, the thread is put back into the scheduler run queue. The process state also gets simultaneously changed to `TASK_RUNNING`, so there is no race condition even if the wake up occurs between the time the task state is set to `TASK_INTERRUPTIBLE` and the time `schedule()` is called. The thread also gets back into the run queue if a `SIGKILL` signal is delivered to it. When the scheduler subsequently picks mykthread from the run queue, execution resumes from Point A.

User Mode Helpers

Mykthread invokes `run_umode_handler()` in Listing 3.1 to notify user space about detected events:

Code View:
```
/* Called from Listing 3.1 */
static void
run_umode_handler(int event_id)
{
  int i = 0;
  char *argv[2], *envp[4], *buffer = NULL;
```

```
  int value;

  argv[i++] = myevent_handler; /* Defined in
                                  kernel/sysctl.c */

  /* Fill in the id corresponding to the data structure
     in trouble */
  if (!(buffer = kmalloc(32, GFP_KERNEL))) return;
  sprintf(buffer, "TROUBLED_DS=%d", event_id);

  /* If no user mode handlers are found, return */
  if (!argv[0]) return; argv[i] = 0;

  /* Prepare the environment for /path/to/helper */
  i = 0;
  envp[i++] = "HOME=/";
  envp[i++] = "PATH=/sbin:/usr/sbin:/bin:/usr/bin";
  envp[i++] = buffer; envp[i]   = 0;

  /* Execute the user mode program, /path/to/helper */
  value = call_usermodehelper(argv[0], argv, envp, 0);

  /* Check return values */
  kfree(buffer);
}
```

The kernel supports a mechanism for requesting user mode programs to help perform certain functions. `run_umode_handler()` uses this facility by invoking `call_usermodehelper()`.

You have to register the user mode program invoked by `run_umode_handler()` via a node in the */proc/sys/* directory. To do so, make sure that `CONFIG_SYSCTL` (files that are part of the */proc/sys/* directory are collectively known as the *sysctl* interface) is enabled in your kernel configuration and add an entry to the `kern_table` array in *kernel/sysctl.c*:

```
{
  .ctl_name     = KERN_MYEVENT_HANDLER, /* Define in
                                           include/linux/sysctl.h */
  .procname     = "myevent_handler",
  .data         = &myevent_handler,
  .maxlen       = 256,
  .mode         = 0644,
  .proc_handler = &proc_dostring,
  .strategy     = &sysctl_string,
},
```

This creates the node */proc/sys/kernel/myevent_handler* in the process filesystem. To register your user mode helper, do the following:

```
bash> echo /path/to/helper > /proc/sys/kernel/myevent_handler
```

This results in */path/to/helper* getting executed when mykthread invokes `run_umode_handler()`.

Mykthread passes the identity of the troubled kernel data structure to the user mode helper through the environment variable `TROUBLED_DS`. The helper can be a simple script like the following that sends you an email alert containing the information it gleaned from its environment:

```
bash> cat /path/to/helper
#!/bin/bash
echo Kernel datastructure $TROUBLED_DS is in trouble | mail -s Alert root
```

`call_usermodehelper()` has to be executed from process context and runs with root privileges. It's implemented using a work queue, which we will soon discuss.

Helper Interfaces

Several useful helper interfaces exist in the kernel to make life easier for device driver developers. One example is the implementation of the doubly linked list library. Many drivers need to maintain and manipulate linked lists of data structures. The kernel's *list* interface routines eliminate the need for chasing list pointers and debugging messy problems related to list maintenance. Let's learn to use helper interfaces such as lists, hlists, work queues, completion functions, notifier blocks, and kthreads.

There are equivalent ways to do what the helper facilities offer. You can, for example, implement your own list manipulation routines instead of using the list library, or use kernel threads to defer work instead of submitting it to work queues. Using standard kernel helper interfaces, however, simplifies your code, weeds out redundancies from the kernel, increases code readability, and helps long-term maintenance.

> Because the kernel is vast, you can always find parts that do not yet take advantage of these helper mechanisms, so updating those code regions might be a good way to start contributing to kernel development.

Linked Lists

To weave doubly linked lists of data structures, use the functions provided in *include/linux/list.h.* Essentially, you embed a `struct list_head` inside your data structure:

```
#include <linux/list.h>

struct list_head {
  struct list_head *next, *prev;
};

struct mydatastructure {
  struct list_head mylist; /* Embed */
  /* ... */                /* Actual Fields */
};
```

`mylist` is the link that chains different instances of `mydatastructure`. If you have multiple `list_head`s embedded inside `mydatastructure`, each of them constitutes a link that renders `mydatastructure` a member of a new list. You can use the list library to add or delete membership from individual lists.

To get the lay of the land before the detail, let's summarize the linked list programming interface offered by the list library. This is done in Table 3.1.

Table 3.1. Linked List Manipulation Functions

Function	Purpose
`INIT_LIST_HEAD()`	Initializes the list head
`list_add()`	Adds an element after the list head

Function	Purpose
`list_add_tail()`	Adds an element to the tail of the list
`list_del()`	Deletes an element from the list
`list_replace()`	Replaces an element in the list with another
`list_entry()`	Loops through all nodes in the list
`list_for_each_entry()/ list_for_each_entry_safe()`	Simpler list iteration interfaces
`list_empty()`	Checks whether there are any elements in the list
`list_splice()`	Joins one list with another

To illustrate list usage, let's implement an example. The example also serves as a foundation to understand the concept of work queues, which is discussed in the next section. Assume that your kernel driver needs to perform a heavy-duty task from an entry point. An example is a task that forces the calling thread to sleep-wait. Naturally, your driver doesn't like to block until the task finishes, because that slows down the responsiveness of applications relying on it. So, whenever the driver needs to perform this expensive work, it defers execution by adding the corresponding routine to a linked list of work functions. The actual work is performed by a kernel thread, which traverses the list and executes the work functions in the background. The driver submits work functions to the tail of the list, while the kernel thread ploughs its way from the head of the list, thus ensuring that work queued first gets done first. Of course, the rest of the driver needs to be designed to suit this scheme of deferred execution. Before understanding this example, however, be aware that we will use the work queue interface in Listing 3.5 to implement the same task in a simpler manner.

Let's first introduce the key driver data structures used by our example:

```
static struct _mydrv_wq {
  struct list_head mydrv_worklist; /* Work List */
  spinlock_t lock;                 /* Protect the list */
  wait_queue_head_t todo;          /* Synchronize submitter
                                      and worker */
} mydrv_wq;


struct _mydrv_work {
  struct list_head mydrv_workitem; /* The work chain */
  void (*worker_func)(void *);     /* Work to perform */
  void *worker_data;               /* Argument to worker_func */
  /* ... */                        /* Other fields */
} mydrv_work;
```

`mydrv_wq` is global to all work submissions. Its members include a pointer to the head of the work list, and a wait queue to communicate between driver functions that submit work and the kernel thread that performs the work. The list helper functions do not protect accesses to list members, so you need to use concurrency mechanisms to serialize simultaneous pointer references. This is done using a spinlock that is also a part of `mydrv_wq`. The driver initialization routine `mydrv_init()` in Listing 3.2 initializes the spinlock, the list head, and the wait queue, and kick starts the worker thread.

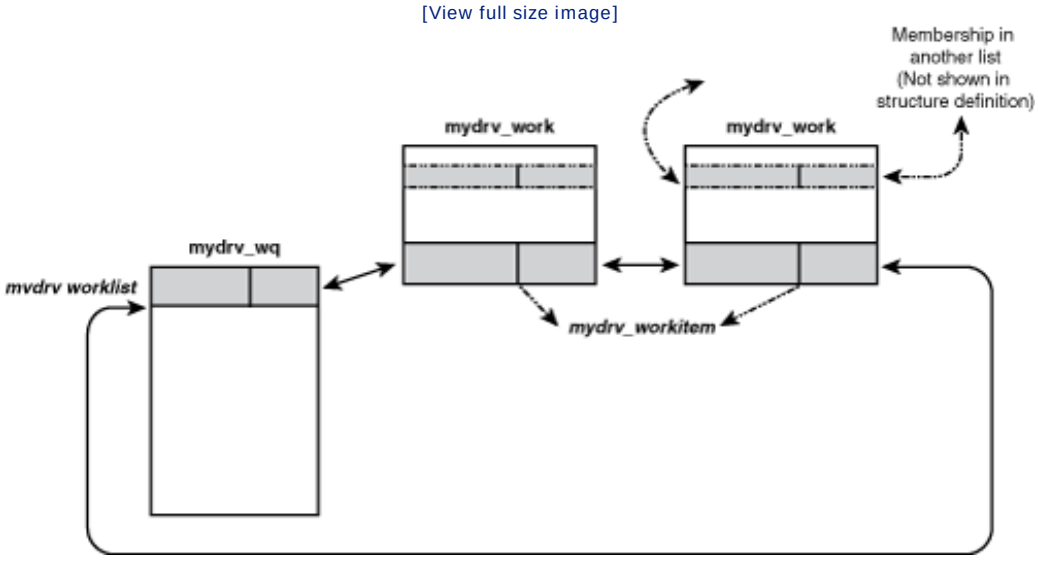
Listing 3.2. Initialize Data Structures

```
static int __init
mydrv_init(void)
{
  /* Initialize the lock to protect against
     concurrent list access */
  spin_lock_init(&mydrv_wq.lock);

  /* Initialize the wait queue for communication
     between the submitter and the worker */
  init_waitqueue_head(&mydrv_wq.todo);

  /* Initialize the list head */
  INIT_LIST_HEAD(&mydrv_wq.mydrv_worklist);

  /* Start the worker thread. See Listing 3.4 */
  kernel_thread(mydrv_worker, NULL,
                CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);
  return 0;
}
```

Before examining the worker thread that executes submitted work, let's look at work submission itself. Listing 3.3 implements a function that other parts of the kernel can use to submit work. It uses `list_add_tail()` to add a work function to the tail of the list. Look at Figure 3.1 to see the physical structure of the work list.

Figure 3.1. Linked list of work functions.

[View full size image]



Listing 3.3. Submitting Work to Be Executed Later

```
int
submit_work(void (*func)(void *data), void *data)
{
  struct _mydrv_work *mydrv_work;

  /* Allocate the work structure */
  mydrv_work = kmalloc(sizeof(struct _mydrv_work), GFP_ATOMIC);
  if (!mydrv_work) return -1;

  /* Populate the work structure */
  mydrv_work->worker_func = func; /* Work function */
  mydrv_work->worker_data = data; /* Argument to pass */
  spin_lock(&mydrv_wq.lock);      /* Protect the list */

  /* Add your work to the tail of the list */
  list_add_tail(&mydrv_work->mydrv_workitem,
                &mydrv_wq.mydrv_worklist);

  /* Wake up the worker thread */
  wake_up(&mydrv_wq.todo);

  spin_unlock(&mydrv_wq.lock);
  return 0;
}
```

To submit a work function `void job(void *)` from a driver entry point, do this:

```
submit_work(job, NULL);
```

After submitting the work function, Listing 3.3 wakes up the worker thread. The general structure of the worker thread shown in Listing 3.4 is similar to standard kernel threads discussed in the previous section. The thread uses `list_entry()` to work its way through all nodes in the list. `list_entry()` returns the container data structure inside which the list node is embedded. Take a closer look at the relevant line in Listing 3.4:

```
mydrv_work = list_entry(mydrv_wq.mydrv_worklist.next,
                        struct _mydrv_work, mydrv_workitem);
```

`mydrv_workitem` is embedded inside `mydrv_work`, so `list_entry()` returns a pointer to the corresponding `mydrv_work` structure. The parameters passed to `list_entry()` are the address of the embedded list node, the type of the container structure, and the field name of the embedded list node.

After executing a submitted work function, the worker thread removes the corresponding node from the list using `list_del()`. Note that `mydrv_wq.lock` is released and reacquired in the time window when the submitted work function is executed. This is because work functions can go to sleep resulting in potential deadlocks if newly scheduled code tries to acquire the same spinlock.

Listing 3.4. The Worker Thread

Code View:

```
static int
mydrv_worker(void *unused)
{

  DECLARE_WAITQUEUE(wait, current);
  void (*worker_func)(void *);
  void *worker_data;
  struct _mydrv_work *mydrv_work;

  set_current_state(TASK_INTERRUPTIBLE);

  /* Spin until asked to die */
  while (!asked_to_die()) {
    add_wait_queue(&mydrv_wq.todo, &wait);

    if (list_empty(&mydrv_wq.mydrv_worklist)) {
      schedule();
      /* Woken up by the submitter */
    } else {
      set_current_state(TASK_RUNNING);
    }
    remove_wait_queue(&mydrv_wq.todo, &wait);

    /* Protect concurrent access to the list */
    spin_lock(&mydrv_wq.lock);

    /* Traverse the list and plough through the work functions
       present in each node */
    while (!list_empty(&mydrv_wq.mydrv_worklist)) {

      /* Get the first entry in the list */
      mydrv_work = list_entry(mydrv_wq.mydrv_worklist.next,
                              struct _mydrv_work, mydrv_workitem);
      worker_func = mydrv_work->worker_func;
      worker_data = mydrv_work->worker_data;

      /* This node has been processed. Throw it
         out of the list */
      list_del(mydrv_wq.mydrv_worklist.next);
      kfree(mydrv_work);   /* Free the node */

      /* Execute the work function in this node */
      spin_unlock(&mydrv_wq.lock);  /* Release lock */
      worker_func(worker_data);
      spin_lock(&mydrv_wq.lock);    /* Re-acquire lock */
    }
    spin_unlock(&mydrv_wq.lock);
    set_current_state(TASK_INTERRUPTIBLE);
  }

  set_current_state(TASK_RUNNING);
  return 0;
}
```

For simplicity, the example code does not perform error handling. For example, if the call to `kernel_thread()` in Listing 3.2 fails, you need to free memory allocated for the corresponding work structure. Also, `asked_to_die()` in Listing 3.4 is left unwritten. It essentially breaks out of the loop if it either detects a delivered signal or receives a communication from the `release()` entry point that the module is about to be unloaded from the kernel.

Before ending this section, let's take a look at another useful list library routine, `list_for_each_entry()`. With this macro, iteration becomes simpler and more readable because you don't have to use `list_entry()` inside the loop. Use the `list_for_each_entry_safe()` variant if you will delete list elements inside the loop. You can replace the following snippet in Listing 3.4:

```
while (!list_empty(&mydrv_wq.mydrv_worklist)) {
  mydrv_work = list_entry(mydrv_wq.mydrv_worklist.next,
                          struct _mydrv_work, mydrv_workitem);
  /* ... */
}
```

with:

```
struct _mydrv_work *temp;
list_for_each_entry_safe(mydrv_work, temp,
                         &mydrv_wq.mydrv_worklist,
                         mydrv_workitem) {
  /* ... */
}
```

You can't use `list_for_each_entry()` in this case because you are removing the entry pointed to by `mydrv_work` inside the loop in Listing 3.4. `list_for_each_entry_safe()` solves this problem using the temporary variable passed as the second argument (`temp`) to save the address of the next entry in the list.

Hash Lists

The doubly linked list implementation discussed previously is not optimal for cases where you want to implement linked data structures such as hash tables. This is because hash tables need only a list head containing a single pointer. To reduce memory overhead for such applications, the kernel provides hash lists (or *hlists*), a variation of lists. Unlike lists, which use the same structure for the list head and list nodes, hlists have separate definitions:

```
struct hlist_head {
  struct hlist_node *first;
};

struct hlist_node {
  struct hlist_node *next, **pprev;
};
```

To suit the scheme of a single-pointer hlist head, the nodes maintain the address of the pointer to the previous node, rather than the pointer itself.

Hash tables are implemented using an array of `hlist_head`s. Each `hlist_head` sources a doubly linked list of `hlist_node`s. A hash function is used to locate the index (or bucket) in the `hlist_head` array. When that is done, you may use hlist helper routines (also defined in *include/linux/list.h*) to operate on the list of `hlist_nodes` linked to the chosen bucket. Look at the implementation of the directory cache (dcache) in

fs/dcache.c for an example.

Work Queues

Work queues are a way to defer work inside the kernel.[2] Deferring work is useful in innumerable situations. Examples include the following:

[2] Softirqs and tasklets are two other mechanisms available for deferring work inside the kernel. Table 4.1 of Chapter 4 compares softirqs, tasklets, and work queues.

- Triggering restart of a network adapter in response to an error interrupt
- Filesystem tasks such as syncing disk buffers
- Sending a command to a disk and following through with the storage protocol state machine

The functionality of work queues is similar to the example described in Listings 3.2 to 3.4. However, work queues can help you accomplish the same task in a simpler manner.

The work queue helper library exposes two interface structures to users: a `workqueue_struct` and a `work_struct`. Follow these steps to use work queues:

1. Create a work queue (or a `workqueue_struct`) with one or more associated kernel threads. To create a kernel thread to service a `workqueue_struct`, use `create_singlethread_workqueue()`. To create one worker thread per CPU in the system, use the `create_workqueue()` variant. The kernel also has default per-CPU worker threads (*events/n*, where *n* is the CPU number) that you can timeshare instead of requesting a separate worker thread. Depending on your application, you might incur a performance hit if you don't have a dedicated worker thread.
2. Create a work element (or a `work_struct`). A `work_struct` is initialized using `INIT_WORK()`, which populates it with the address and argument of your work function.
3. Submit the work element to the work queue. A `work_struct` can be submitted to a dedicated queue using `queue_work()`, or to the default kernel worker thread using `schedule_work()`.

Let's rewrite Listings 3.2 to 3.4 to take advantage of the work queue interface. This is done in Listing 3.5. The entire kernel thread, as well as the spinlock and the wait queue, vanish inside the work queue interface. Even the call to `create_singlethread_workqueue()` goes away if you are using the default kernel worker thread.

Listing 3.5. Using Work Queues to Defer Work

Code View:

```
#include <linux/workqueue.h>

struct workqueue_struct *wq;

/* Driver Initialization */
static int __init
mydrv_init(void)
{
  /* ... */
  wq = create_singlethread_workqueue("mydrv");
  return 0;
}
  /* Work Submission. The first argument is the work function, and
     the second argument is the argument to the work function */
  int
  submit_work(void (*func)(void *data), void *data)
  {
    struct work_struct *hardwork;

    hardwork = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    /* Init the work structure */
    INIT_WORK(hardwork, func, data);

    /* Enqueue Work */
    queue_work(wq, hardwork);
    return 0;
}
```

If you are using work queues, you will get linker errors unless you declare your module as licensed under GPL. This is because the kernel exports these functions only to GPLed code. If you look at the kernel work queue implementation, you will see this restriction expressed in statements such as this:

```
EXPORT_SYMBOL_GPL(queue_work);
```

To announce that your module is *copyleft*-ed under GPL, declare the following:

```
MODULE_LICENSE("GPL");
```

Notifier Chains

Notifier chains are used to send status change messages to code regions that request them. Unlike hard-coded mechanisms, notifiers offer a versatile technique for getting alerted when events of interest are generated. Notifiers were originally added for passing network events to concerned sections of the kernel but are now used for many other purposes. The kernel implements predefined notifiers for significant events. Examples of such notifications include the following:

- Die notification, which is sent when a kernel function triggers a trap or a fault, caused by an "oops," page fault, or a breakpoint hit. If you are, for example, writing a device driver for a medical grade card, you might want to register yourself with the die notifier so that you can attempt to turn off the medical electronics if a kernel panic occurs.

- Net device notification, which is generated when a network interface goes up or down.

- CPU frequency notification, which is dispatched when there is a transition in the processor frequency.

- Internet address notification, which is sent when a change is detected in the IP address of a network interface.

An example user of notifiers is the *High-level Data Link Control* (HDLC) protocol driver *drivers/net/wan/hdlc.c*, which registers itself with the net device notifier chain to sense carrier changes.

To attach your code to a notifier chain, you have to register an event handler with the associated chain. An event identifier and a notifier-specific argument are passed as arguments to the handler routine when the concerned event is generated. To define a custom notifier chain, you have to additionally implement the infrastructure to ignite the chain when the event is detected.

Listing 3.6 contains examples of using predefined and user-defined notifiers. Table 3.2 contains a brief description of the notifier chains used by Listing 3.6 and the events they propagate, so look at the listing and the table in tandem.

Table 3.2. Notifier Chains and the Events They Propagate

Notifier Chain	Description

Die Notifier Chain (`die_chain`)	`my_die_event_handler()` attaches to the die notifier chain, `die_chain`, using `register_die_notifier()`. To trigger invocation of `my_die_event_handler()`, introduce an invalid dereference somewhere in your code, such as the following:
	`int *q = 0;`
	`*q = 1;`
	
	When this code snippet is executed, `my_die_event_handler()` gets called, and you will see a message like this:
	
	`my_die_event_handler: OOPs! at EIP=f00350e7`
	
	The die event notifier passes the `die_args` structure to the registered event handler. This argument contains a pointer to the `regs` structure, which carries a snapshot of processor register contents when the fault occurred. `my_die_event_handler()` prints the contents of the instruction pointer register.

Netdevice Notifier Chain(`netdev_chain`)	`my_dev_event_handler()` attaches to the net device notifier chain, `netdev_chain`, using `register_netdevice_notifier()`. You can generate this event by changing the state of a network interface such as Ethernet (`ethX`) or loopback (`lo`): **`bash> ifconfig eth0 up`**

Notifier Chain	Description
	This results in the execution of `my_dev_event_handler()`. The handler is passed a pointer to `struct net_device` as argument, which contains the name of the network interface. `my_dev_event_handler()` uses this information to produce the following message: `my_dev_event_handler: Val=1, Interface=eth0` `Val=1` corresponds to the `NETDEV_UP` event as defined in *include/linux/notifier.h*.

User-Defined Notifier Chain	Listing 3.6 also implements a user-defined notifier chain, `my_noti_chain`. Assume that you want an event to be generated whenever a user reads from a particular file in the process filesystem. Add the following in the associated procfs `read` routine:
	`blocking_notifier_call_chain(&my_noti_chain, 100, NULL);`
	
	This results in the invocation of `my_event_handler()` whenever you read from the corresponding */proc* file and results in the following message:
	
	`my_event_handler: Val=100`
	
	`Val` contains the identity of the generated event, which is `100` for this example. The function argument is left unused.

You have to unregister event handlers from notifier chains when your module is released from the kernel. For example, if you *up* or *down* a network interface after unloading the code in Listing 3.6, you will be rankled by an "oops," unless you perform an `unregister_netdevice_notifier(&my_dev_notifier)` from the module's `release()` method. This is because the notifier chain continues to think that the handler code is valid, even though it has been pulled out of the kernel.

Listing 3.6. Notifier Event Handlers

Code View:

```
#include <linux/notifier.h>
#include <asm/kdebug.h>
#include <linux/netdevice.h>
#include <linux/inetdevice.h>

/* Die Notifier Definition */
static struct notifier_block my_die_notifier = {
  .notifier_call = my_die_event_handler,
};
/* Die notification event handler */
int
my_die_event_handler(struct notifier_block *self,
                         unsigned long val, void *data)
{
  struct die_args *args = (struct die_args *)data;

  if (val == 1) { /* '1' corresponds to an "oops" */
    printk("my_die_event: OOPs! at EIP=%lx\n", args->regs->eip);
```

```
  } /* else ignore */
  return 0;
}


/* Net Device notifier definition */
static struct notifier_block my_dev_notifier = {
  .notifier_call = my_dev_event_handler,
};


/* Net Device notification event handler */
int my_dev_event_handler(struct notifier_block *self,
                              unsigned long val, void *data)
{
  printk("my_dev_event: Val=%ld, Interface=%s\n", val,
           ((struct net_device *) data)->name);
  return 0;
}


/* User-defined notifier chain implementation */
static BLOCKING_NOTIFIER_HEAD(my_noti_chain);

static struct notifier_block my_notifier = {
  .notifier_call = my_event_handler,
};

/* User-defined notification event handler */
int my_event_handler(struct notifier_block *self,
                         unsigned long val, void *data)
{
  printk("my_event: Val=%ld\n", val);
  return 0;
}
/* Driver Initialization */
static int __init
my_init(void)
{
  /* ... */

  /* Register Die Notifier */
  register_die_notifier(&my_die_notifier);

  /* Register Net Device Notifier */
  register_netdevice_notifier(&my_dev_notifier);

  /* Register a user-defined Notifier */
  blocking_notifier_chain_register(&my_noti_chain, &my_notifier);

  /* ... */
}
```

`my_noti_chain` in Listing 3.6 is declared as a blocking notifier using `BLOCKING_NOTIFIER_HEAD()` and is registered via `blocking_notifier_chain_register()`. This means that the notifier handler is always invoked from process context. So, the handler function, `my_event_handler()`, is allowed to go to sleep. If your notifier

handler can be called from interrupt context, declare the notifier chain using `ATOMIC_NOTIFIER_HEAD()`, and register it via `atomic_notifier_chain_register()`.

> **The Old Notifier Interface**
>
> Kernel releases earlier than 2.6.17 supported only a general-purpose notifier chain. The notifier registration function `notifier_chain_register()` was internally protected using a spinlock, but the routine that walked the notifier chain dispatching events to notifier handlers (`notifier_call_chain()`) was lockless. The lack of locking was because of the possibility that the handler functions may go to sleep, unregister themselves while running, or get called from interrupt context. The lockless implementation introduced race conditions, however. The new notifier API is built over the original interface and is intended to overcome its limitations.

Completion Interface

Many parts of the kernel initiate certain activities as separate execution threads and then wait for them to complete. The *completion* interface is an efficient and easy way to implement such code patterns.

Some example usage scenarios include the following:

- Your driver module is assisted by a kernel thread. If you rmmod the module, the `release()` method is invoked before removing the module code from kernel space. The release routine asks the thread to kill itself and blocks until the thread completes its exit. Listing 3.7 implements this case.

- You are writing a portion of a block device driver (discussed in Chapter 14, "Block Drivers") that queues a read request to a device. This triggers a state machine change implemented as a separate thread or work queue. The driver wants to wait until the operation completes before proceeding with another activity. Look at *drivers/block/floppy.c* for an example.

- An application requests an *Analog-to-Digital Converter* (ADC) driver for a data sample. The driver initiates a conversion request waits, until an interrupt signals completion of conversion, and returns the data.

Listing 3.7. Synchronizing Using Completion Functions

Code View:

```
static DECLARE_COMPLETION(my_thread_exit);      /* Completion */
static DECLARE_WAIT_QUEUE_HEAD(my_thread_wait); /* Wait Queue */
int pink_slip = 0;                              /* Exit Flag */

/* Helper thread */
static int
my_thread(void *unused)
{
  DECLARE_WAITQUEUE(wait, current);

  daemonize("my_thread");
  add_wait_queue(&my_thread_wait, &wait);

  while (1) {
    /* Relinquish processor until event occurs */
```

```
    set_current_state(TASK_INTERRUPTIBLE);
    schedule();
    /* Control gets here when the thread is woken
       up from the my_thread_wait wait queue */

    /* Quit if let go */
    if (pink_slip) {
      break;
    }
    /* Do the real work */
    /* ... */

  }

  /* Bail out of the wait queue */
  __set_current_state(TASK_RUNNING);
  remove_wait_queue(&my_thread_wait, &wait);

  /* Atomically signal completion and exit */
  complete_and_exit(&my_thread_exit, 0);
}

/* Module Initialization */
static int __init
my_init(void)
{
  /* ... */

  /* Kick start the thread */
  kernel_thread(my_thread, NULL,
                CLONE_FS | CLONE_FILES | CLONE_SIGHAND | SIGCHLD);

  /* ... */
}

/* Module Release */
static void __exit
my_release(void)
{
  /* ... */
  pink_slip = 1;                        /* my_thread must go */
  wake_up(&my_thread_wait);             /* Activate my_thread */
  wait_for_completion(&my_thread_exit); /* Wait until my_thread
                                           quits */
  /* ... */
}
```

A completion object can be declared statically using `DECLARE_COMPLETION()` or created dynamically with `init_completion()`. A thread can signal completion with the help of `complete()` or `complete_all()`. A caller can wait for completion via `wait_for_completion()`.

In Listing 3.7, `my_release()` raises an exit request flag by setting `pink_slip` before waking up `my_thread()`. It then calls `wait_for_completion()` to wait until `my_thread()` completes its exit. `my_thread()`, on its part, wakes up to find `pink_slip` set, and does the following:

1. Signals completion to `my_release()`

2. Kills itself

`my_thread()` accomplishes these two steps atomically using `complete_and_exit()`. Using `complete_and_exit()` shuts the window between module exit and thread exit that opens if you separately invoke `complete()` and `exit()`.

We will use the completion API when we develop an example telemetry driver in Chapter 11.

Kthread Helpers

Kthread helpers add a coating over the raw thread creation routines and simplify the task of thread management.

Listing 3.8 rewrites Listing 3.7 using the kthread helper interface. `my_init()` now uses `kthread_create()` rather than `kernel_thread()`. You can pass the thread's name to `kthread_create()` rather than explicitly call `daemonize()` within the thread.

The kthread interface provides you free access to a built-in exit synchronization mechanism implemented using the completion interface. So, as `my_release()` does in Listing 3.8, you may directly call `kthread_stop()` instead of laboriously setting `pink_slip`, waking up `my_thread()`, and waiting for it to complete using `wait_for_completion()`. Similarly, `my_thread()` can make a neat call to `kthread_should_stop()` to check whether it ought to call it a day.

Listing 3.8. Synchronizing Using Kthread Helpers

Code View:

```
/* '+' and '-' show the differences from Listing 3.7 */

#include <linux/kthread.h>

/* Assistant Thread */
static int
my_thread(void *unused)
{
   DECLARE_WAITQUEUE(wait, current);
-   daemonize("my_thread");

-   while (1) {
+   /* Continue work if no other thread has
+    * invoked kthread_stop() */
+   while (!kthread_should_stop()) {
      /* ... */
-     /* Quit if let go */
-     if (pink_slip) {
-       break;
-     }
      /* ... */
    }
    __set_current_state(TASK_RUNNING);
    remove_wait_queue(&my_thread_wait, &wait);
```

```
-   complete_and_exit(&my_thread_exit, 0);
+   return 0;
 }

+   struct task_struct *my_task;

 /* Module Initialization */
 static int __init
 my_init(void)
 {
    /* ... */
-   kernel_thread(my_thread, NULL,
-                 CLONE_FS | CLONE_FILES | CLONE_SIGHAND |
-                 SIGCHLD);
+   my_task = kthread_create(my_thread, NULL, "%s", "my_thread");
+   if (my_task) wake_up_process(my_task);

    /* ... */
 }

 /* Module Release */
 static void __exit
 my_release(void)
 {
    /* ... */
-   pink_slip = 1;
-   wake_up(&my_thread_wait);
-   wait_for_completion(&my_thread_exit);
+   kthread_stop(my_task);

    /* ... */
 }
```

Instead of creating the thread using `kthread_create()` and activating it via `wake_up_process()` as done in Listing 3.8, you may use the following single call:

```
kthread_run(my_thread, NULL, "%s", "my_thread");
```

Error-Handling Aids

Several kernel functions return pointer values. Whereas callers usually check for failure by comparing the return value with `NULL`, they typically need more information to decipher the exact nature of the error that has occurred. Because kernel addresses have redundant bits, they can be overloaded to encode error semantics. This is done with the help of a set of helper routines. Listing 3.9 implements a simple usage example.

Listing 3.9. Using Error-Handling Aids

Code View:

```
#include <linux/err.h>

char *
collect_data(char *userbuffer)
{

  char *buffer;

  /* ... */
  buffer = kmalloc(100, GFP_KERNEL);
  if (!buffer) { /* Out of memory */
    return ERR_PTR(-ENOMEM);
  }

  /* ... */
  if (copy_from_user(buffer, userbuffer, 100)) {
    return ERR_PTR(-EFAULT);
  }
  /* ... */

  return(buffer);
}


int
my_function(char *userbuffer)
{
  char *buf;

  /* ... */
  buf = collect_data(userbuffer);
  if (IS_ERR(buf)) {
    printk("Error returned is %d!\n", PTR_ERR(buf));
  }
  /* ... */

}
```

If `kmalloc()` fails inside `collect_data()` in Listing 3.9, you will get the following message:

```
Error returned is -12!
```

However, if `collect_data()` is successful, it returns a valid pointer to a data buffer. As another example, let's add error handling using `IS_ERR()` and `PTR_ERR()` to the thread creation code in Listing 3.8:

```
   my_task = kthread_create(my_thread, NULL, "%s", "mydrv");

+  if (!IS_ERR(my_task)) {
+    /* Success */
     wake_up_process(my_task);
+  } else {
+    /* Failure */
```

```
+    printk("Error value returned=%d\n", PTR_ERR(my_task));
+  }
```

Looking at the Sources

The ksoftirqd, pdflush, and khubd kernel threads live in *kernel/softirq.c*, *mm/pdflush.c*, and *drivers/usb/core/hub.c*, respectively.

The `daemonize()` function can be found in *kernel/exit.c.* For the implementation of user mode helpers, look at *kernel/kmod.c*.

The list and hlist library routines reside in *include/linux/list.h*. They are used all over the kernel, so you will find usage examples in most subdirectories. An example is the `request_queue` structure defined in *include/linux/blkdev.h*, which holds a linked list of disk I/O requests. We look at this data structure in Chapter 14.

Go to www.ussg.iu.edu/hypermail/linux/kernel/0007.3/0805.html and follow the discussion thread in the mailing list for an interesting debate between Linus Torvalds and Andi Kleen about the pros and cons of complementing the list library with hlist helper routines.

The kernel work queue implementation lives in *kernel/workqueue.c.* To understand the real-world use of work queues, look at the PRO/Wireless 2200 network driver, *drivers/net/wireless/ipw2200.c*.

The kernel notifier chain implementation lives in *kernel/sys.c* and *include/linux/notifier.h.* Look at *kernel/sched.c* and *include/linux/completion.h* for the guts of the completion interface. *kernel/kthread.c* contains the source code for kthread helpers, and *include/linux/err.h* includes definitions of error handling aids.

Table 3.3 contains a summary of the main data structures used in this chapter and the location of their definitions in the source tree. Table 3.4 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 3.3. Summary of Data Structures

Data Structure	Location	Description
`wait_queue_t`	*include/linux/wait.h*	Used by threads that desire to wait for an event or a system resource
`list_head`	*include/linux/list.h*	Kernel structure to weave a doubly linked list of data structures
`hlist_head`	*include/linux/list.h*	Kernel structure to implement hash tables
`work_struct`	*include/linux/workqueue.h*	Implements work queues, which are a way to defer work inside the kernel
`notifier_block`	*include/linux/notifier.h*	Implements notifier chains, which are used to send status changes to code regions that request them
`completion`	*include/linux/completion.h*	Used to initiate activities as separate threads and then wait for them to complete

Table 3.4. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`DECLARE_WAITQUEUE()`	*include/linux/wait.h*	Declares a wait queue.
`add_wait_queue()`	*kernel/wait.c*	Queues a task to a wait queue. The task goes to sleep until it's woken up by another thread or interrupt handler.
`remove_wait_queue()`	*kernel/wait.c*	Dequeues a task from a wait queue.
`wake_up_interruptible()`	*include/linux/wait.h* *kernel/sched.c*	Wakes up a task sleeping inside a wait queue and puts it back into the scheduler run queue.
`schedule()`	*kernel/sched.c*	Relinquishes the processor and allows other parts of the kernel to run.
`set_current_state()`	*include/linux/sched.h*	Sets the run state of a process. The state can be one of `TASK_RUNNING`, `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_STOPPED`, `TASK_TRACED`, `EXIT_ZOMBIE`, or `EXIT_DEAD`.
`kernel_thread()`	*arch/your-arch/kernel/process.c*	Creates a kernel thread.
`daemonize()`	*kernel/exit.c*	Activates a kernel thread without attaching user resources and changes the parent of the calling thread to kthreadd.
`allow_signal()`	*kernel/exit.c*	Enables delivery of a specified signal.
`signal_pending()`	*include/linux/sched.h*	Checks whether a signal has been delivered. There are no signal handlers inside the kernel, so you have to explicitly check whether a signal has been delivered.
`call_usermodehelper()`	*include/linux/kmod.h* *kernel/kmod.c*	Executes a user mode program.
`Linked list library functions`	*include/linux/list.h*	See Table 3.1.
`register_die_notifier()`	*arch/your-arch/kernel/traps.c*	Registers a die notifier.
`register_netdevice_notifier()`	*net/core/dev.c*	Registers a netdevice notifier.
`register_inetaddr_notifier()`	*net/ipv4/devinet.c*	Registers an inetaddr notifier.
`BLOCKING_NOTIFIER_HEAD()`	*include/linux/notifier.h*	Creates a user-defined blocking notifier.
`blocking_notifier_chain_register()`	*kernel/sys.c*	Registers a blocking notifier.

Kernel Interface	Location	Description
`blocking_notifier_call_chain()`	*kernel/sys.c*	Dispatches an event to a blocking notifier chain.
`ATOMIC_NOTIFIER_HEAD()`	*include/linux/notifier.h*	Creates an atomic notifier.
`atomic_notifier_chain_register()`	*kernel/sys.c*	Registers an atomic notifier.
`DECLARE_COMPLETION()`	*include/linux/completion.h*	Statically declares a completion object.
`init_completion()`	*include/linux/completion.h*	Dynamically declares a completion object.
`complete()`	*kernel/sched.c*	Announces completion.
`wait_for_completion()`	*kernel/sched.c*	Waits until the completion object completes.
`complete_and_exit()`	*kernel/exit.c*	Atomically signals completion and exit.
`kthread_create()`	*kernel/kthread.c*	Creates a kernel thread.
`kthread_stop()`	*kernel/kthread.c*	Asks a kernel thread to stop.
`kthread_should_stop()`	*kernel/kthread.c*	A kernel thread can poll on this function to detect whether another thread has asked it to stop via `kthread_stop()`.
`IS_ERR()`	*include/linux/err.h*	Finds out whether the return value is an error code.

Chapter 4. Laying the Groundwork

In This Chapter

- Introducing Devices and Drivers 90
- Interrupt Handling 92
- The Linux Device Model 103
- Memory Barriers 114
- Power Management 114
- Looking at the Sources 115

We are now within whispering distance of writing a device driver. Before doing that, however, let's equip ourselves with some driver concepts. We start the chapter by getting an idea of the book's problem statement; we will look at the typical devices and I/O interfaces present on PC-compatible systems and embedded computers. Interrupt handling is an integral part of most drivers, so we next cover the art of writing interrupt handlers. We then turn our attention to the new device model introduced in the 2.6 kernel. The new model is built around abstractions such as *sysfs*, *kobjects*, *device classes*, and *udev*, which distill commonalities from device drivers. The new device model also weeds policies out of kernel space and pushes them to user space, resulting in a total revamp of features such as */dev* node management, hotplug, coldplug, module autoload, and firmware download.

Introducing Devices and Drivers

User applications cannot directly communicate with hardware because that entails possessing privileges such as executing special instructions and handling interrupts. Device drivers assume the burden of interacting with hardware and export interfaces that applications and the rest of the kernel can use to access devices. Applications operate on devices via nodes in the */dev* directory and glean device information using nodes in the */sys* directory.[1]

[1] As you'll learn later, networking applications route their requests to the underlying driver using a different mechanism.

Figure 4.1 shows the hardware block diagram of a typical PC-compatible system. As you can see, the system supports diverse devices and interface technologies such as memory, video, audio, USB, PCI, WiFi, PCMCIA, I^2C, IDE, Ethernet, serial port, keyboard, mouse, floppy drive, parallel port, and Infrared. The memory controller and the graphics controller are part of a *North Bridge* chipset in the PC architecture, whereas peripheral buses are sourced out of a *South Bridge*.

Figure 4.1. Hardware block diagram of a PC-compatible system.

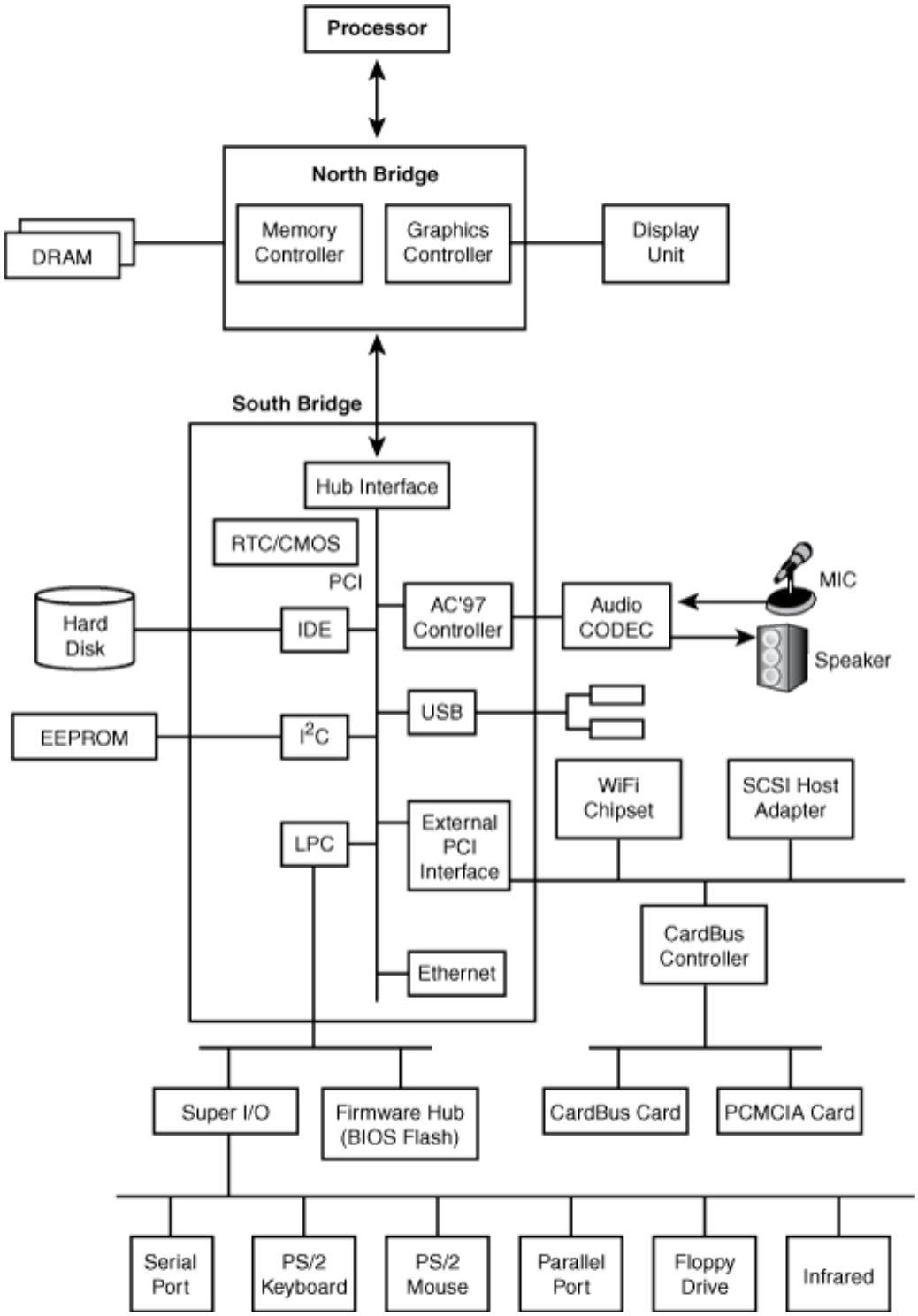
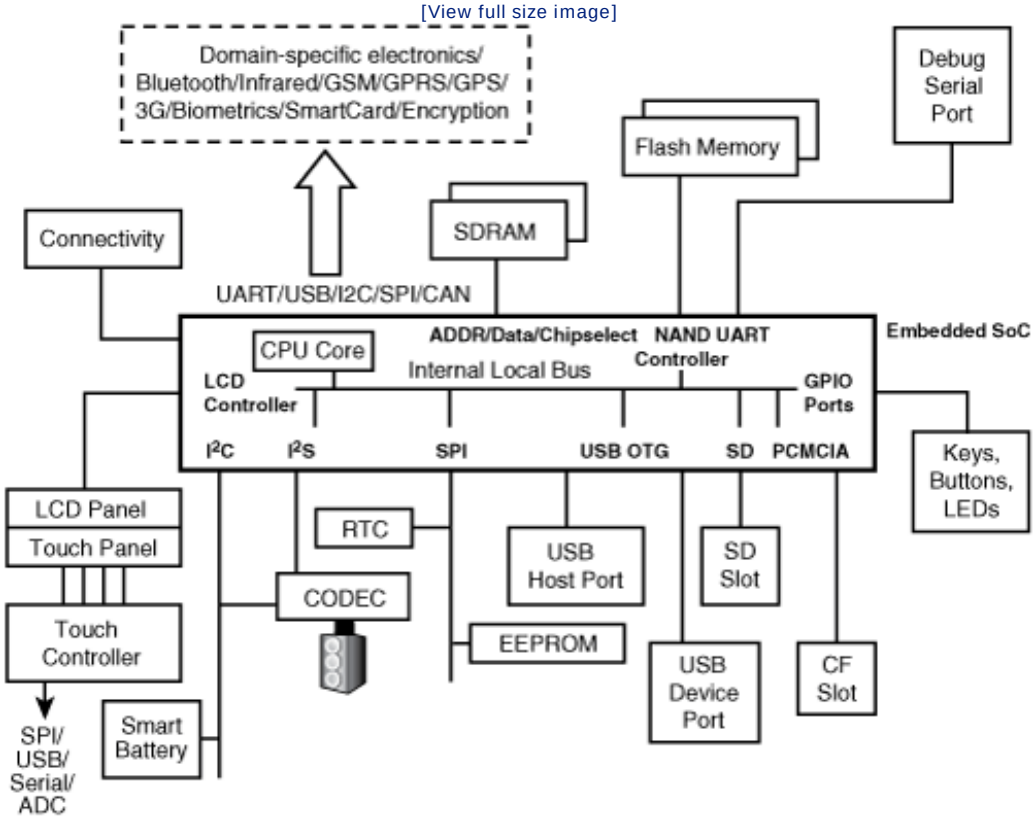


Figure 4.2 illustrates a similar block diagram for a hypothetical embedded device. This diagram contains several interfaces not typical in the PC world such as flash memory, LCD, touch screen, and cellular modem.

Figure 4.2. Hardware block diagram of an embedded system.

[View full size image]



Naturally, the capability to access peripheral devices is a crucial part of a system's functioning. Device drivers provide the engine to achieve this. The rest of the chapters in this book will zoom in on a device interface and teach you how to implement the corresponding device driver.

Chapter 4. Laying the Groundwork

In This Chapter

- Introducing Devices and Drivers 90
- Interrupt Handling 92
- The Linux Device Model 103
- Memory Barriers 114
- Power Management 114
- Looking at the Sources 115

We are now within whispering distance of writing a device driver. Before doing that, however, let's equip ourselves with some driver concepts. We start the chapter by getting an idea of the book's problem statement; we will look at the typical devices and I/O interfaces present on PC-compatible systems and embedded computers. Interrupt handling is an integral part of most drivers, so we next cover the art of writing interrupt handlers. We then turn our attention to the new device model introduced in the 2.6 kernel. The new model is built around abstractions such as *sysfs*, *kobjects*, *device classes*, and *udev*, which distill commonalities from device drivers. The new device model also weeds policies out of kernel space and pushes them to user space, resulting in a total revamp of features such as */dev* node management, hotplug, coldplug, module autoload, and firmware download.

Introducing Devices and Drivers

User applications cannot directly communicate with hardware because that entails possessing privileges such as executing special instructions and handling interrupts. Device drivers assume the burden of interacting with hardware and export interfaces that applications and the rest of the kernel can use to access devices. Applications operate on devices via nodes in the */dev* directory and glean device information using nodes in the */sys* directory.[1]

[1] As you'll learn later, networking applications route their requests to the underlying driver using a different mechanism.

Figure 4.1 shows the hardware block diagram of a typical PC-compatible system. As you can see, the system supports diverse devices and interface technologies such as memory, video, audio, USB, PCI, WiFi, PCMCIA, I^2C, IDE, Ethernet, serial port, keyboard, mouse, floppy drive, parallel port, and Infrared. The memory controller and the graphics controller are part of a *North Bridge* chipset in the PC architecture, whereas peripheral buses are sourced out of a *South Bridge*.

Figure 4.1. Hardware block diagram of a PC-compatible system.

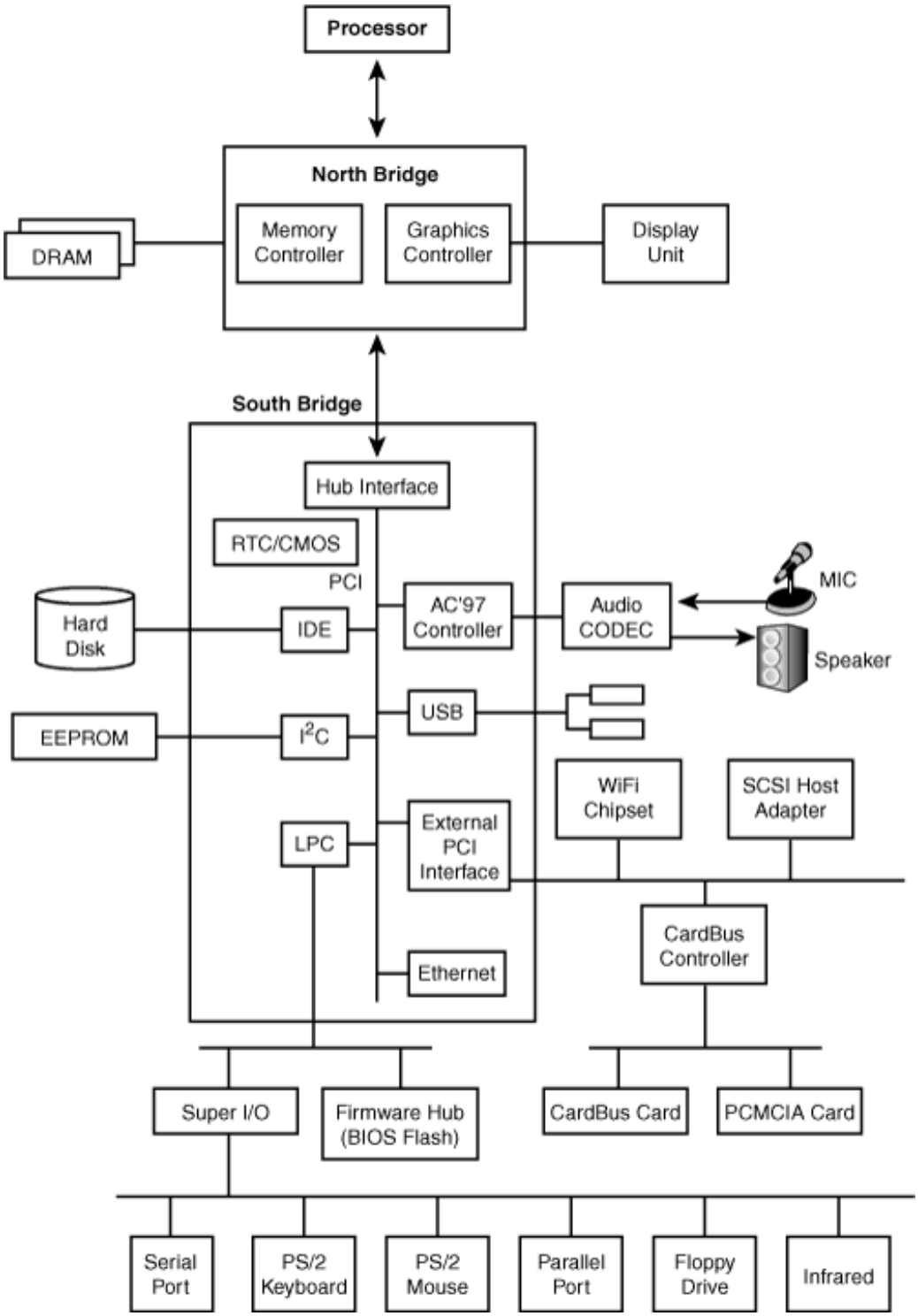
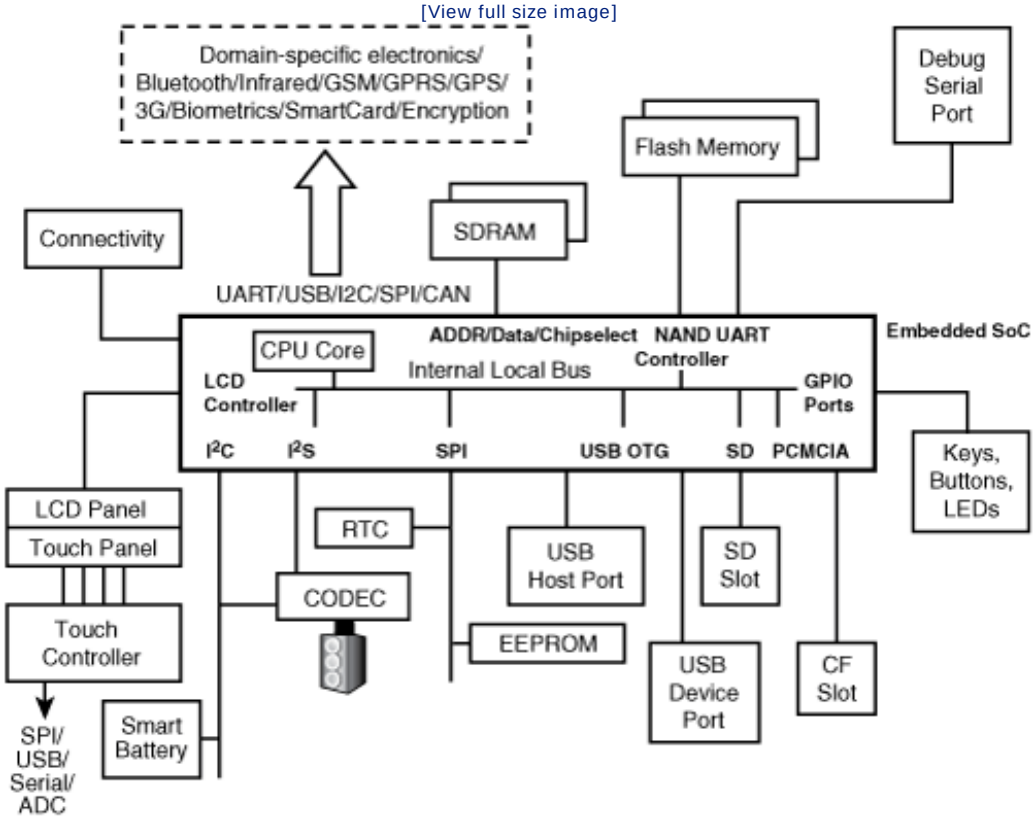


Figure 4.2 illustrates a similar block diagram for a hypothetical embedded device. This diagram contains several interfaces not typical in the PC world such as flash memory, LCD, touch screen, and cellular modem.

Figure 4.2. Hardware block diagram of an embedded system.

[View full size image]



Naturally, the capability to access peripheral devices is a crucial part of a system's functioning. Device drivers provide the engine to achieve this. The rest of the chapters in this book will zoom in on a device interface and teach you how to implement the corresponding device driver.

Interrupt Handling

Because of the indeterminate nature of I/O, and speed mismatches between I/O devices and the processor, devices request the processor's attention by asserting certain hardware signals asynchronously. These hardware signals are called *interrupts.* Each interrupting device is assigned an associated identifier called an *interrupt request* (IRQ) number. When the processor detects that an interrupt has been generated on an IRQ, it abruptly stops what it's doing and invokes an *interrupt service routine* (ISR) registered for the corresponding IRQ. *Interrupt handlers* (ISRs) execute in interrupt context.

Interrupt Context

ISRs are critical pieces of code that directly converse with the hardware. They are given the privilege of instant execution in the larger interest of system performance. However, if ISRs are not quick and lightweight, they contradict their own philosophy. VIPs are given preferential treatment, but it's incumbent on them to minimize the resulting inconvenience to the public. To compensate for rudely interrupting the current thread of execution, ISRs have to politely execute in a restricted environment called interrupt context (or *atomic context*).

Here is a list of do's and don'ts for code executing in interrupt context:

1. It's a jailable offense if your interrupt context code goes to sleep. Interrupt handlers cannot relinquish the processor by calling sleepy functions such as `schedule_timeout()`. Before invoking a kernel API from your interrupt handler, penetrate the nested invocation train and ensure that it does not internally trigger a blocking wait. For example, `input_register_device()` looks harmless from the surface, but tosses a call to `kmalloc()` under the hood specifying `GFP_KERNEL` as an argument. As you saw in Chapter 2, "A Peek Inside the Kernel," if your system's free memory dips below a watermark, `kmalloc()` sleep-waits for memory to get freed up by the swapper, if you invoke it in this manner.

2. For protecting critical sections inside interrupt handlers, you can't use mutexes because they may go to sleep. Use spinlocks instead, and use them only if you must.

3. Interrupt handlers cannot directly exchange data with user space because they are not connected to user land via process contexts. This brings us to another reason why interrupt handlers cannot sleep: The scheduler works at the granularity of processes, so if interrupt handlers sleep and are scheduled out, how can they be put back into the run queue?

4. Interrupt handlers are supposed to get out of the way quickly but are expected to get the job done. To circumvent this Catch-22, interrupt handlers usually split their work into two. The slim *top half* of the handler flags an acknowledgment claiming that it has serviced the interrupt but, in reality, offloads all the hard work to a fat *bottom half.* Execution of the bottom half is deferred to a later point in time when all interrupts are enabled. You will learn to develop bottom halves while discussing *softirqs* and *tasklets* later.

5. You need not design interrupt handlers to be reentrant. When an interrupt handler is running, the corresponding IRQ is disabled until the handler returns. So, unlike process context code, different instances of the same handler will not run simultaneously on multiple processors.

6. Interrupt handlers can be interrupted by handlers associated with IRQs that have higher priority. You can prevent this nested interruption by specifically requesting the kernel to treat your interrupt handler as a

fast handler. Fast handlers run with all interrupts disabled on the local processor. Before disabling interrupts or labeling your interrupt handler as fast, be aware that interrupt-off times are bad for system performance. More the interrupt-off times, more is the interrupt latency, or the delay before a generated interrupt is serviced. Interrupt latency is inversely proportional to the real time responsiveness of the system.

A function can check the value returned by `in_interrupt()` to find out whether it's executing in interrupt context.

Unlike asynchronous interrupts generated by external hardware, there are classes of interrupts that arrive synchronously. Synchronous interrupts are so called because they don't occur unexpectedly—the processor itself generates them by executing an instruction. Both external and synchronous interrupts are handled by the kernel using identical mechanisms.

Examples of synchronous interrupts include the following:

- Exceptions, which are used to report grave runtime errors
- Software interrupts such as the `int 0x80` instruction used to implement system calls on the x86 architecture

Assigning IRQs

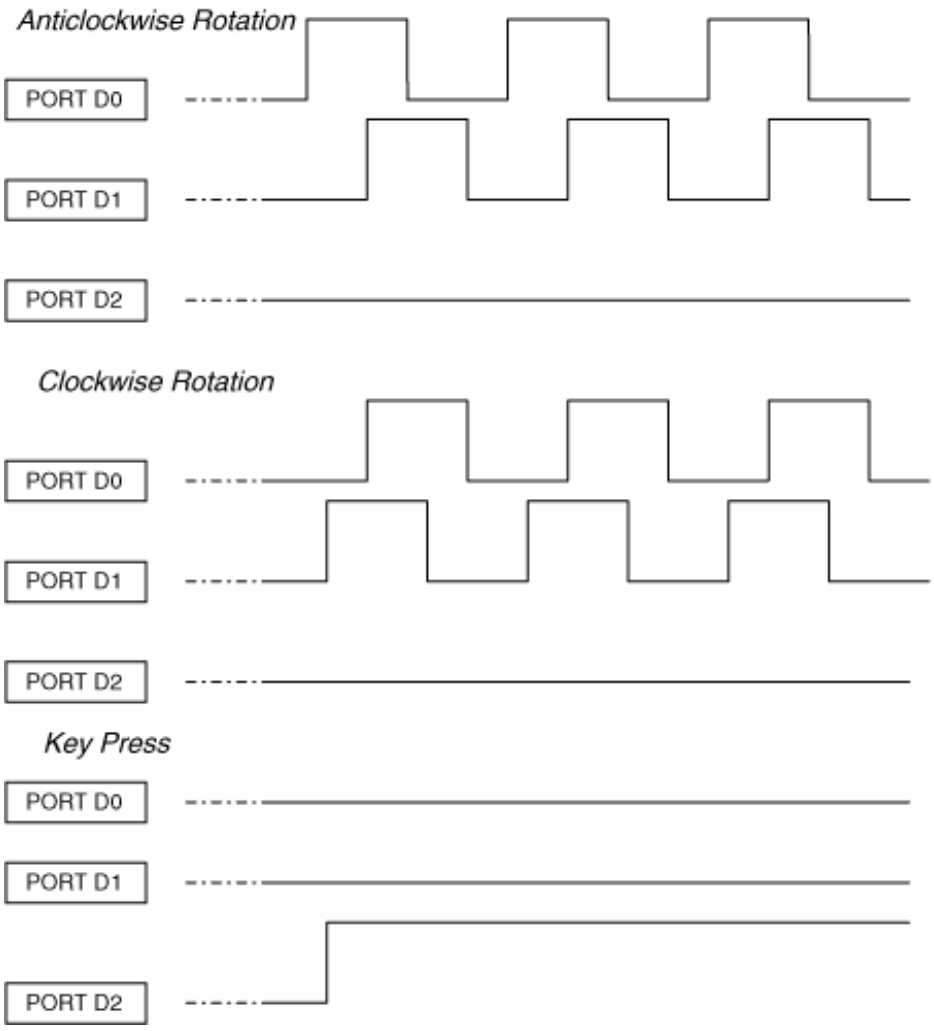
Device drivers have to connect their IRQ number to an interrupt handler. For this, they need to know the IRQ assigned to the device they're driving. IRQ assignments can be straightforward or may require complex probing. In the PC architecture, for example, timer interrupts are assigned IRQ 0, and RTC interrupts answer to IRQ 8. Modern bus technologies such as PCI are sophisticated enough to respond to queries regarding their IRQs (assigned by the BIOS when it walks the bus during boot). PCI drivers can poke into earmarked regions in the device's configuration space and figure out the IRQ. For older devices such as *Industries Standard Architecture* (ISA)-based cards, the driver might have to leverage hardware-specific knowledge to probe and decipher the IRQ.

Take a look at */proc/interrupts* for a list of active IRQs on your system.

Device Example: Roller Wheel

Now that you have learned the basics of interrupt handling, let's implement an interrupt handler for an example roller wheel device. Roller wheels can be found on some phones and PDAs for easy menu navigation and are capable of three movements: clockwise rotation, anticlockwise rotation, and key-press. Our imaginary roller wheel is wired such that any of these movements interrupt the processor on IRQ 7. Three low order bits of *General Purpose I/O* (GPIO) Port D of the processor are connected to the roller device. The waveforms generated on these pins corresponding to different wheel movements are shown in Figure 4.3. The job of the interrupt handler is to decipher the wheel movements by looking at the Port D GPIO data register.

Figure 4.3. Sample wave forms generated by the roller wheel.



The driver has to first request the IRQ and associate an interrupt handler with it:

```
#define ROLLER_IRQ  7
static irqreturn_t roller_interrupt(int irq, void *dev_id);


if (request_irq(ROLLER_IRQ, roller_interrupt, IRQF_DISABLED |
                IRQF_TRIGGER_RISING, "roll", NULL)) {
  printk(KERN_ERR "Roll: Can't register IRQ %d\n", ROLLER_IRQ);
  return -EIO;
}
```

Let's look at the arguments passed to `request_irq()`. The IRQ number is not queried or probed but hard-coded to `ROLLER_IRQ` in this simple case as per the hardware connection. The second argument, `roller_interrupt()`, is the interrupt handler routine. Its prototype specifies a return type of `irqreturn_t`, which can be `IRQ_HANDLED` if the interrupt is handled successfully or `IRQ_NONE` if it isn't. The return value assumes more significance for I/O technologies such as PCI, where multiple devices can share the same IRQ.

The `IRQF_DISABLED` flag specifies that this interrupt handler has to be treated as a fast handler, so the kernel

has to disable interrupts while invoking the handler. `IRQF_TRIGGER_RISING` announces that the roller wheel generates a rising edge on the interrupt line when it wants to signal an interrupt. In other words, the roller wheel is an *edge-sensitive* device. Some devices are instead *level-sensitive* and keep the interrupt line asserted until the CPU services it. To flag an interrupt as level-sensitive, use the `IRQF_TRIGGER_HIGH` flag. Other possible values for this argument include `IRQF_SAMPLE_RANDOM` (used in the section, "Pseudo Char Drivers" in Chapter 5, "Character Drivers") and `IRQF_SHARED` (used to specify that this IRQ is shared among multiple devices).

The next argument, `"roll"`, is used to identify this device in data generated by files such as */proc/interrupts.* The final parameter, set to `NULL` in this case, is relevant only for shared interrupt handlers and is used to identify each device sharing the IRQ line.

> Starting with the 2.6.19 kernel, there have been some changes to the interrupt handler interface. Interrupt handlers used to take a third argument (`struct pt_regs *`) that contained a pointer to CPU registers, but this has been removed starting with the 2.6.19 kernel. Also, the `IRQF_xxx` family of interrupt flags replaced the `SA_xxx` family. For example, with earlier kernels, you had to use `SA_INTERRUPT` rather than `IRQF_DISABLED` to mark an interrupt handler as fast.

Driver initialization is not a good place for requesting an IRQ because that can hog that valuable resource even when the device is not in use. So, device drivers usually request the IRQ when the device is opened by an application. Similarly, the IRQ is freed when the application closes the device and not while exiting the driver module. Freeing an IRQ is done as follows:

```
free_irq(int irq, void *dev_id);
```

Listing 4.1 shows the implementation of the roller interrupt handler. `roller_interrupt()` takes two arguments: the IRQ and the device identifier passed as the final argument to the associated `request_irq()`. Look at Figure 4.3 side by side with this listing.

Listing 4.1. The Roller Interrupt Handler

Code View:

```
spinlock_t roller_lock = SPIN_LOCK_UNLOCKED;
static DECLARE_WAIT_QUEUE_HEAD(roller_poll);

static irqreturn_t
roller_interrupt(int irq, void *dev_id)
{
  int i, PA_t, PA_delta_t, movement = 0;

  /* Get the waveforms from bits 0, 1 and 2
     of Port D as shown in Figure 4.3 */
  PA_t = PORTD & 0x07;

  /* Wait until the state of the pins change.
     (Add some timeout to the loop) */
  for (i=0; (PA_t==PA_delta_t); i++){
    PA_delta_t = PORTD & 0x07;
  }

  movement = determine_movement(PA_t, PA_delta_t); /* See below */
```

```
  spin_lock(&roller_lock);

  /* Store the wheel movement in a buffer for
     later access by the read()/poll() entry points */
  store_movements(movement);

  spin_unlock(&roller_lock);

  /* Wake up the poll entry point that might have
     gone to sleep, waiting for a wheel movement */
  wake_up_interruptible(&roller_poll);

  return IRQ_HANDLED;
}
int
determine_movement(int PA_t, int PA_delta_t)
{
  switch (PA_t){
    case 0:
      switch (PA_delta_t){
      case 1:
        movement = ANTICLOCKWISE;
        break;
      case 2:
        movement = CLOCKWISE;
        break;
      case 4:
        movement = KEYPRESSED;
        break;
      }
      break;
    case 1:
      switch (PA_delta_t){
      case 3:
        movement = ANTICLOCKWISE;
        break;
      case 0:
        movement = CLOCKWISE;
        break;
      }
      break;
    case 2:
      switch (PA_delta_t){
      case 0:
        movement = ANTICLOCKWISE;
        break;
      case 3:
        movement = CLOCKWISE;
        break;
      }
      break;
    case 3:
      switch (PA_delta_t){
      case 2:
        movement = ANTICLOCKWISE;
        break;
      case 1:
        movement = CLOCKWISE;
        break;
```

```
      }
    case 4:
      movement = KEYPRESSED;
      break;
  }
}
```

Driver entry points such as `read()` and `poll()` operate in tandem with `roller_interrupt()`. For example, when the handler deciphers wheel movement, it wakes up any waiting `poll()` threads that may have gone to sleep in response to a `select()` system call issued by an application such as X Windows. Revisit Listing 4.1 and implement the complete roller driver after learning the internals of character drivers in Chapter 5.

Listing 7.3 in Chapter 7, "Input Drivers," takes advantage of the kernel's *input* interface to convert this roller wheel into a roller mouse.

Let's end this section by introducing some functions that enable and disable interrupts on a particular IRQ. `enable_irq(ROLLER_IRQ)` enables interrupt generation when the roller wheel moves, while `disable_irq(ROLLER_IRQ)` does the reverse. `disable_irq_nosync(ROLLER_IRQ)` disables roller interrupts but does not wait for any currently executing instance of `roller_interrupt()` to return. This `nosync` flavor of `disable_irq()` is faster but can potentially cause race conditions. Use this only when you know that there can be no races. An example user of `disable_irq_nosync()` is *drivers/ide/ide-io.c*, which blocks interrupts during initialization, because some systems have trouble with that.

Softirqs and Tasklets

As discussed previously, interrupt handlers have two conflicting requirements: They are responsible for the bulk of device data processing, but they have to exit as fast as possible. To bail out of this situation, interrupt handlers are designed in two parts: a hurried and harried top half that interacts with the hardware, and a relaxed bottom half that does most of the processing with all interrupts enabled. Unlike interrupts, bottom halves are synchronous because the kernel decides when to execute them. The following mechanisms are available in the kernel to defer work to a bottom half: softirqs, tasklets, and work queues.

Softirqs are the basic bottom half mechanism and have strong locking requirements. They are used only by a few performance-sensitive subsystems such as the networking layer, SCSI layer, and kernel timers. Tasklets are built on top of softirqs and are easier to use. It's recommended to use tasklets unless you have crucial scalability or speed requirements. A primary difference between a softirq and a tasklet is that the former is reentrant whereas the latter isn't. Different instances of a softirq can run simultaneously on different processors, but that is not the case with tasklets.

To illustrate the usage of softirqs and tasklets, assume that the roller wheel in the previous example has inherent hardware problems due to the presence of moving parts (say, the wheel gets stuck occasionally) resulting in the generation of out-of-spec waveforms. A stuck wheel can continuously generate spurious interrupts and potentially freeze the system. To get around this problem, capture the wave stream, run some analysis on it, and dynamically switch from interrupt mode to a polled mode if the wheel looks stuck, and vice versa if it's unstuck. Capture the wave stream from the interrupt handler and perform the analysis from a bottom half. Listing 4.2 implements this using softirqs, and Listing 4.3 uses tasklets. Both are simplified variants of Listing 4.1. This reduces the handler to two functions: `roller_capture()` that obtains a wave snippet from GPIO Port D, and `roller_analyze()` that runs an algorithmic analysis on the wave and switches to polled mode if required.

Listing 4.2. Using Softirqs to Offload Work from Interrupt Handlers

Code View:

```
void __init
roller_init()
{
  /* ... */

  /* Open the softirq. Add an entry for ROLLER_SOFT_IRQ in
     the enum list in include/linux/interrupt.h */
  open_softirq(ROLLER_SOFT_IRQ, roller_analyze, NULL);
}


/* The bottom half */
void
roller_analyze()
{
  /* Analyze the waveforms and switch to polled mode if required */
}
/* The interrupt handler */
static irqreturn_t
roller_interrupt(int irq, void *dev_id)
{
  /* Capture the wave stream */
  roller_capture();

  /* Mark softirq as pending */
  raise_softirq(ROLLER_SOFT_IRQ);

  return IRQ_HANDLED;
}
```

To define a softirq, you have to statically add an entry to *include/linux/interrupt.h.* You can't define one dynamically. `raise_softirq()` announces that the corresponding softirq is pending execution. The kernel will execute it at the next available opportunity. This can be during exit from an interrupt handler or via the *ksoftirqd* kernel thread.

Listing 4.3. Using Tasklets to Offload Work from Interrupt Handlers

Code View:

```
struct roller_device_struct { /* Device-specific structure */
  /* ... */
  struct tasklet_struct tsklt;
  /* ... */
}

void __init roller_init()
{
  struct roller_device_struct *dev_struct;
  /* ... */

  /* Initialize tasklet */
  tasklet_init(&dev_struct->tsklt, roller_analyze, dev);
}


/* The bottom half */
void
roller_analyze()
{
/* Analyze the waveforms and switch to
   polled mode if required */
}
/* The interrupt handler */
static irqreturn_t
roller_interrupt(int irq, void *dev_id)
{
  struct roller_device_struct *dev_struct;

  /* Capture the wave stream */
  roller_capture();

  /* Mark tasklet as pending */
  tasklet_schedule(&dev_struct->tsklt);

  return IRQ_HANDLED;
}
```

`tasklet_init()` dynamically initializes a tasklet. The function does not allocate memory for a `tasklet_struct`, rather you have to pass the address of an allocated one. `tasklet_schedule()` announces that the corresponding tasklet is pending execution. Like for interrupts, the kernel offers a bunch of functions to control the execution state of tasklets on systems having multiple processors:

- `tasklet_enable()` enables tasklets.
- `tasklet_disable()` disables tasklets and waits until any currently executing tasklet instance has exited.
- `tasklet_disable_nosync()` has semantics similar to `disable_irq_nosync()`. The function does not wait for active instances of the tasklet to finish execution.

You have seen the differences between interrupt handlers and bottom halves, but there are a few similarities, too. Interrupt handlers and tasklets are both not reentrant. And neither of them can go to sleep. Also, interrupt handlers, tasklets, and softirqs cannot be preempted.

Work queues are a third way to defer work from interrupt handlers. They execute in process context and are allowed to sleep, so they can use drowsy functions such as mutexes. We discussed work queues in the preceding chapter when we looked at various kernel helper facilities. Table 4.1 compares softirqs, tasklets, and work queues.

Table 4.1. Comparing Softirqs, Tasklets, and Work Queues

	Softirqs	Tasklets	Work Queues
Execution context	Deferred work runs in interrupt context.	Deferred work runs in interrupt context.	Deferred work runs in process context.
Reentrancy	Can run simultaneously on different CPUs.	Cannot run simultaneously on different CPUs. Different CPUs can run different tasklets, however.	Can run simultaneously on different CPUs.
Sleep semantics	Cannot go to sleep.	Cannot go to sleep.	May go to sleep.
Preemption	Cannot be preempted/scheduled.	Cannot be preempted/scheduled.	May be preempted/scheduled.
Ease of use	Not easy to use.	Easy to use.	Easy to use.
When to use	If deferred work will not go to sleep and if you have crucial scalability or speed requirements.	If deferred work will not go to sleep.	If deferred work may go to sleep.

There is an ongoing debate in LKML on the feasibility of getting rid of the tasklet interface. Tasklets enjoy more priority than process context code, so they present latency problems. Moreover, as you learned, they are constrained not to sleep and to execute on the same CPU. It's being suggested that all existing tasklets be converted to softirqs or work queues on a case-by-case basis.

The `-rt` patch-set alluded to in Chapter 2 moves interrupt handling to kernel threads to achieve wider preemption coverage.

The Linux Device Model

The new Linux device model introduces C++-like abstractions that factor out commonalities from device drivers into bus and core layers. Let's look at the different components constituting the device model such as *udev*, *sysfs*, *kobjects*, and *device classes* and their effects on key kernel subsystems such as */dev* node management, hotplug, firmware download, and module autoload. *Udev* is the best vantage point to view the benefits of the device model, so let's start from there.

Udev

Years ago when Linux was young, it was not fun to administer device nodes. All the needed nodes (which could run into thousands) had to be statically created under the */dev* directory. This problem, in fact, dated all the way back to original UNIX systems. With the advent of the 2.4 kernels came *devfs*, which introduced dynamic device node creation. Devfs provided services to generate device nodes in an in-memory filesystem, but the onus of naming the nodes still rested with device drivers. Device naming policy is administrative and does not mix well with the kernel, however. The place for policy is in header files, kernel module parameters, or user space. Udev arrived on the scene to push device management to user space.

Udev depends on the following to do its work:

1. Kernel *sysfs* support, which is an important part of the Linux device model. Sysfs is an in-memory filesystem mounted under */sys* at boot time (look at */etc/fstab* for the specifier). We will look at sysfs in the next section, but for now, take the corresponding sysfs file accesses for granted.

2. A set of user-space daemons and utilities such as *udevd* and *udevinfo*.

3. User-specified rules located in the */etc/udev/rules.d/* directory. You may frame rules to get a consistent view of your devices.

To understand how to use udev, let's look at an example. Assume that you have a USB DVD drive and a USB CD-RW drive. Depending on the order in which you hotplug these devices, one of them is assigned the name */dev/sr0*, and the other gets the name */dev/sr1.* During pre-udev days, you had to figure out the associated names before you could use the devices. But with udev, you can consistently view the DVD (as say, */dev/usbdvd)* and the CD-RW (as say, */dev/usbcdrw*) irrespective of the order in which they are plugged in or out.

First, pull product attributes from corresponding files in sysfs. Assume that the (Targus) DVD drive has been assigned the device node */dev/sr0* and that the (Addonics) CD-RW drive has been given the name */dev/sr1*. Use `udevinfo` to collect device information:

Code View:
```
bash> udevinfo -a -p /sys/block/sr0
...
looking at the device chain at
'/sys/devices/pci0000:00/0000:00:1d.7/usb1/1-4':
 BUS=»usb»
 ID=»1-4»
 SYSFS{bConfigurationValue}=»1»
 ...
```

```
 SYSFS{idProduct}=»0701»
 SYSFS{idVendor}=»05e3»
 SYSFS{manufacturer}=»Genesyslogic»
 SYSFS{maxchild}=»0»
 SYSFS{product}=»USB Mass Storage Device»
 ...

bash> udevinfo -a -p /sys/block/sr1
 ...
 looking at the device chain at
 '/sys/devices/pci0000:00/0000:00:1d.7/usb1/1-3':
 BUS=»usb»
 ID=»1-3»
 SYSFS{bConfigurationValue}=»2»
  ...
 SYSFS{idProduct}=»0302»
 SYSFS{idVendor}=»0dbf»
 SYSFS{manufacturer}=»Addonics»
 SYSFS{maxchild}=»0»
 SYSFS{product}=»USB to IDE Cable»
 ...
```

Next, let's use the product information gleaned to identify the devices and add udev naming rules. Create a file called */etc/udev/rules.d/40-cdvd.rules* and add the following rules to it:

```
BUS="usb", SYSFS{idProduct}="0701", SYSFS{idVendor}="05e3",
KERNEL="sr[0-9]*", NAME="%k", SYMLINK="usbdvd"

BUS="usb", SYSFS{idProduct}="0302", SYSFS{idVendor}="0dbf",
KERNEL="sr[0-9]*", NAME="%k", SYMLINK="usbcdrw"
```

The first rule tells udev that whenever it finds a USB device with a product ID of 0x0701, vendor ID of 0x05e3, and a name starting with *sr*, it should create a node of the same name under */dev* and produce a symbolic link named *usbdvd* to the created node. Similarly, the second rule orders creation of a symbolic link named *usbcdrw* for the CD-RW drive.

To test for syntax errors in your rules, run udevtest on */sys/block/sr**. To turn on verbose messages in */var/log/messages*, set `udev_log` to "yes" in */etc/udev/udev.conf*. To repopulate the */dev* directory with newly added rules on-the-fly, restart udev using udevstart. When this is done, your DVD drive consistently appears to the system as */dev/usbdvd*, and your CD-RW drive always appears as */dev/usbcdrw.* You can deterministically mount them from shell scripts using commands such as these:

```
mount /dev/usbdvd /mnt/dvd
```

Consistent naming of device nodes (and network interfaces) is not the sole capability of udev. It has metamorphed into the Linux hotplug manager, too. Udev is also in charge of automatically loading modules on demand and downloading microcode onto devices that need them. But before digging into those capabilities, let's obtain a basic understanding of the innards of the device model.

Sysfs, Kobjects, and Device Classes

Sysfs, kobjects, and device classes are the building blocks of the device model but are publicity shy and prefer to remain behind the scenes. They are mostly in the usage domain of bus and core implementations, and hide inside APIs that provide services to device drivers.

Sysfs is the user-space manifestation of the kernel's structured device model. It's similar to procfs in that both are in-memory filesystems containing information about kernel data structures. Whereas procfs is a generic window into kernel internals, sysfs is specific to the device model. Sysfs is, hence, not a replacement for procfs. Information such as process descriptors and sysctl parameters belong to procfs and not sysfs. As will be apparent soon, udev depends on sysfs for most of its extended functions.

Kobjects introduce an encapsulation of common object properties such as usage reference counts. They are usually embedded within larger structures. The following are the main fields of a kobject, which is defined in *include/linux/kobject.h*:

1. A *kref* object that performs reference count management. The `kref_init()` interface initializes a kref, `kref_get()` increments the reference count associated with the kref, and `kref_put()` decrements the reference count and frees the object if there are no remaining references. The URB structure (explained in Chapter 11, "Universal Serial Bus"), for example, contains a kref to track the number of references to it.[2]

   [2] The `usb_alloc_urb()` interface calls `kref_init()`, `usb_submit_urb()` invokes `kref_get()`, and `usb_free_urb()` calls `kref_put()`.

2. A pointer to a *kset*, which is an object set to which the kobject belongs.

3. A *kobj_type*, which is an object type that describes the kobject.

Kobjects are intertwined with sysfs. Every kobject instantiated within the kernel has a sysfs representation.

The concept of device classes is another feature of the device model and is an interface you're more likely to use in a driver. The class interface abstracts the idea that each device falls under a broader class (or category) of devices. A USB mouse, a PS/2 keyboard, and a joystick all fall under the *input* class and own entries under */sys/class/input/*.

Figure 4.4 shows the sysfs hierarchy on a laptop that has an external USB mouse connected to it. The top-level *bus*, *class*, and *device* directories are expanded to show that sysfs provides a view of the USB mouse based on its device type as well as its physical connection. The mouse is an input class device but is physically a USB device answering to two endpoint addresses, a control endpoint *ep00*, and an interrupt endpoint, *ep81.* The USB port in question belongs to the USB host controller on bus 2, and the USB host controller itself is bridged to the CPU via the PCI bus. If these details are not making much sense at this point, don't worry; rewind to this section after reading the chapters that teach input drivers (Chapter 7), PCI drivers (Chapter 10, "Peripheral Component Interconnect"), and USB drivers (Chapter 11).

Figure 4.4. Sysfs hierarchy of a USB mouse.

Code View:

```
[/sys]
     +[block]
     -[bus]—[usb]—[devices]—[usb2]—[2-2]—[2-2:1.0]-[usbendpoint:usbdev2.2-ep81]
     -[class]-[input]—[mouse2]—[device]—[bus]—[usbendpoint:usbdev2.2-ep81]
             -[usb_device]—[usbdev2.2]—[device]—[bus]
             -[usb_endpoint]—[usbdev2.2-ep00]—[device]
                            —[usbdev2.2-ep81]—[device]
     -[devices]—[pci0000:00]—[0000:00:1d:1]—[usb2]—[2-2]—[2-2:1.0]
     +[firmware]
     +[fs]
     +[kernel]
     +[module]
     +[power]
```

Browse through */sys* looking for entries that associate with another device (for example, your network card) to get a better feel of its hierarchical organization. The section "Addressing and Identification" in Chapter 10 illustrates how sysfs mirrors the physical connection of a CardBus Ethernet-Modem card on a laptop.

The class programming interface is built on top of kobjects and sysfs, so it's a good place to start digging to understand the end-to-end interactions between the components of the device model. Let's turn to the RTC driver for an example. The RTC driver (*drivers/char/rtc.c*) is a miscellaneous (or "misc") driver. We discuss misc drivers in detail when we look at character device drivers in Chapter 5.

Insert the RTC driver module and look at the nodes created under */sys* and */dev:*

```
bash> modprobe rtc
bash> ls -lR /sys/class/misc
drwr-xr-x 2 root root 0 Jan 15 01:23 rtc
/sys/class/misc/rtc:
total 0
-r--r--r-- 1 root root 4096 Jan 15 01:23 dev
--w------- 1 root root 4096 Jan 15 01:23 uevent
bash> ls -l /dev/rtc
crw-r--r-- 1 root root 10, 135 Jan 15 01:23 /dev/rtc
```

/sys/class/misc/rtc/dev contains the major and minor numbers (discussed in the next chapter) assigned to this device, */sys/class/misc/rtc/uevent* is used for coldplugging (discussed in the next section), and */dev/rtc* is used by applications to access the RTC driver.

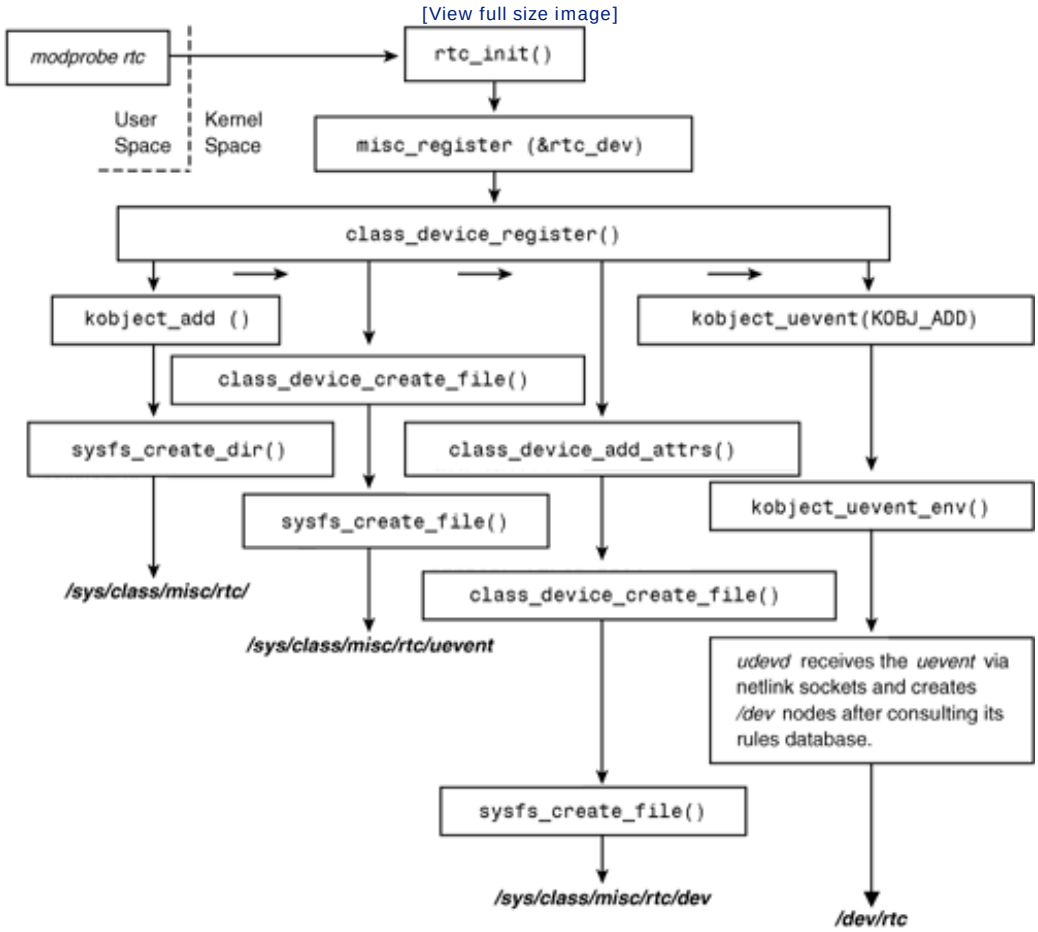
Let's understand the code flow through the device model. Misc drivers utilize the services of `misc_register()` during initialization, which looks like this if you peel off some code:

```
/* ... */
dev = MKDEV(MISC_MAJOR, misc->minor);

misc->class = class_device_create(misc_class, NULL, dev,
                                  misc->dev,
                                  "%s", misc->name);
if (IS_ERR(misc->class)) {
  err = PTR_ERR(misc->class);
  goto out;
```

```
}
/* ... */
```

Figure 4.5 continues to peel off more layers to get to the bottom of the device modeling. It illustrates the transitions that ripple through classes, kobjects, sysfs, and udev, which result in the generation of the */sys* and */dev* files listed previously.

Figure 4.5. Tying the pieces of the device model.



Look at the parallel port LED driver (Listing 5.6 in the section "Talking to the Parallel Port" in Chapter 5) and the virtual mouse input driver (Listing 7.2 in the section "Device Example: Virtual Mouse" in Chapter 7) for examples on creating device control files inside sysfs.

Another abstraction that is part of the device model is the *bus-device-driver* programming interface. Kernel device support is cleanly structured into buses, devices, and drivers. This renders the individual driver implementations simpler and more general. *Bus* implementations can, for example, search for drivers that can handle a particular device.

Consider the kernel's I^2C subsystem (explored in Chapter 8, "The Inter-Integrated Circuit Protocol"). The I^2C layer consists of a core infrastructure, device drivers for bus adapters, and drivers for client devices. The I^2C

core layer registers each detected I^{2}C bus adapter using `bus_register()`. When an I^{2}C client device (say, an *Electrically Erasable Programmable Read-Only Memory* [EEPROM] chip) is probed and detected, its existence is recorded via `device_register()`. Finally, the I^{2}C EEPROM client driver registers itself using `driver_register()`. These registrations are performed indirectly using service functions offered by the I^{2}C core.

`bus_register()` adds a corresponding entry to */sys/bus/*, while `device_register()` adds entries under */sys/devices/*. `struct bus_type`, `struct device`, and `struct device_driver` are the main data structures used respectively by buses, devices, and drivers. Take a peek inside *include/linux/device.h* for their definitions.

Hotplug and Coldplug

Devices connected to a running system on-the-fly are said to be *hotplugged*, whereas those connected prior to system boot are considered to be *coldplugged.* Earlier, the kernel used to notify user space about hotplug events by invoking a helper program registered via the */proc* filesystem. But when current kernels detect hotplug, they dispatch uevents to user space via netlink sockets. Netlink sockets are an efficient mechanism to communicate between kernel space and user space using socket APIs. At the user-space end, udevd, the daemon that manages device node creation and removal, receives the uevents and manages hotplug.

To see how hotplug handling has evolved recently, let's consider progressive levels of udev running different versions of the 2.6 kernel:

1. With a *udev-039* package and a 2.6.9 kernel, when the kernel detects a hotplug event, it invokes the user space helper registered with */proc/sys/kernel/hotplug*. This defaults to */sbin/hotplug*, which receives attributes of the hotplugged device in its environment. */sbin/hotplug* looks inside the hotplug configuration directory (usually */etc/hotplug.d/default/*) and runs, for example, */etc/hotplug.d/default/10-udev.hotplug*, after executing other scripts under */etc/hotplug/*.

   ```
   bash> ls -l /etc/hotplug.d/default/
   ...
   lrwcrwxrwx 1 root root 14 May 11 2005 10-udev.hotplug -> /sbin/udevsend
   ...
   ```

   When */sbin/udevsend* thus gets executed, it passes the hotplugged device information to udevd.

2. With udev-058 and a 2.6.11 kernel, the story changes somewhat. The udevsend utility replaces */sbin/hotplug*:

   ```
   bash> cat /proc/sys/kernel/hotplug
   /sbin/udevsend
   ```

3. With the latest levels of udev and the kernel, udevd assumes full responsibility of managing hotplug without depending on udevsend. It now pulls hotplug events directly from the kernel via netlink sockets (see Figure 4.4). */proc/sys/kernel/hotplug* contains nothing:

   ```
   bash> cat /proc/sys/kernel/hotplug
   bash>
   ```

Udev also handles coldplug. Because udev is part of user space and is started only after the kernel boots, a special mechanism is needed to emulate hotplug events over coldplugged devices. At boot time, the kernel creates a file named *uevent* under sysfs for all devices and emits coldplug events to those files. When udev starts, it reads all the *uevent* files from */sys* and generates hotplug uevents for each coldplugged device.

Microcode Download

You have to feed microcode to some devices before they can get ready for action. The microcode gets executed by an on-card microcontroller. Device drivers used to store microcode inside static arrays in header files. But this has become untenable because microcode is usually distributed as proprietary binary images by device vendors, and that doesn't mix homogeneously with the GPL-ed kernel. Another reason against mixing firmware with kernel sources is that they run on different release time lines. The solution apparently is to separately maintain microcode in user space and pass it down to the kernel when required. Sysfs and udev provide an infrastructure to achieve this.

Let's take the example of the Intel PRO/Wireless 2100 WiFi mini PCI card found on several laptops. The card is built around a microcontroller that needs to execute externally supplied microcode for normal operation. Let's walk through the steps that the Linux driver follows to download microcode to the card. Assume that you have obtained the required microcode image (*ipw2100-1.3.fw*) from http://ipw2100.sourceforge.net/firmware.php and saved it under */lib/firmware/* on your system and that you have inserted the driver module *ipw2100.ko*:

1. During initialization, the driver invokes the following:

   ```
   request_firmware(..,"ipw2100-1.3.fw",..);
   ```

2. This dispatches a hotplug uevent to user space, along with the identity of the requested microcode image.

3. Udevd receives the uevent and responds by invoking */sbin/firmware_helper*. For this, it uses a rule similar to the following from a file under */etc/udev/rules.d/*:

   ```
   ACTION=="add", SUBSYSTEM=="firmware", RUN="/sbin/firmware_helper"
   ```

4. */sbin/firmware_helper* looks inside */lib/firmware/* and locates the requested microcode image *ipw2100-1.3.fw*. It dumps the image to */sys/class/0000:02:02.0/data*. (0000:02:02 is the PCI *bus:device:function* identifier of the WiFi card in this case.)

5. The driver receives the microcode and downloads it onto the device. When done, it calls `release_firmware()` to free the corresponding data structures.

6. The driver goes through the rest of the initializations and the WiFi adapter beacons.

Module Autoload

Automatically loading kernel modules on demand is a convenient feature that Linux supports. To understand how the kernel emits a "module fault" and how udev handles it, let's insert a Xircom CardBus Ethernet adapter into a laptop's PC Card slot:

1. During compile time, the identity of supported devices is generated as part of the driver module object. Take a peek at the driver that supports the Xircom CardBus Ethernet combo card (*drivers/net/tulip/xircom_cb.c*) and find this snippet:

```
static struct pci_device_id xircom_pci_table[] = {
    {0x115D, 0x0003, PCI_ANY_ID, PCI_ANY_ID,},
    {0,},
};

/* Mark the device table */
MODULE_DEVICE_TABLE(pci, xircom_pci_table);
```

   This declares that the driver can support any card having a PCI vendor ID of `0x115D` and a PCI device ID of `0x0003` (more on this in Chapter 10). When you install the driver module, the depmod utility looks inside the module image and deciphers the IDs present in the device table. It then adds the following entry to */lib/modules/kernel-version/modules.alias*:

```
alias pci:v0000115Dd00000003sv*sd*bc*sc*i* xircom_cb
```

   where `v` stands for VendorID, `d` for DeviceID, `sv` for subvendorID, and `*` for wildcard match.

2. When you hotplug the Xircom card into a CardBus slot, the kernel generates a uevent that announces the identity of the newly inserted device. You may look at the generated uevent using udevmonitor:

```
bash> udevmonitor --env
   ...
   MODALIAS=pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00
   ...
```

3. Udevd receives the uevent via a netlink socket and invokes modprobe with the above MODALIAS that the kernel passed up to it:

```
modprobe pci:v0000115Dd00000003sv0000115Dsd00001181bc02sc00i00
```

4. Modprobe finds the matching entry in */lib/modules/kernel-version/modules.alias* created during Step 1, and proceeds to insert *xircom_cb:*

```
bash> lsmod
Module      Size   Used by
xircom_cb   10433  0
...
```

The card is now ready to surf.

You may want to revisit this section after reading Chapter 10.

Udev on Embedded Devices

One school of thought deprecates the use of udev in favor of statically created device nodes on embedded devices for the following reasons:

- Udev creates */dev* nodes during each reboot, compared to static nodes that are created only once during software install. If your embedded device uses flash storage, flash pages that hold */dev* nodes suffer an erase-write cycle on each boot in the case of the former, and this reduces flash life span. (Flash memory is discussed in detail in Chapter 17, "Memory Technology Devices.") You do have the option of mounting */dev* over a RAM-based filesystem, however.

- Udev contributes to increased boot time.

- Udev features such as dynamic creation of */dev* nodes and autoloading of modules create a degree of indeterminism that some solution designers prefer to avoid on special-purpose embedded devices, especially ones that do not interact with the outside world via hotpluggable buses. According to this point of view, static node creation and boot-time insertion of any modules provide more control over the system and make it easier to test.

Memory Barriers

Many processors and compilers reorder instructions to achieve optimal execution speeds. The reordering is done such that the new instruction stream is semantically equivalent to the original one. However, if you are, for example, writing to memory mapped registers on an I/O device, instruction reordering can generate unexpected side effects. To prevent the processor from reordering instructions, you can insert a barrier in your code. The `wmb()` function inserts a road block that prevents writes from moving through it, `rmb()` provides a read barricade that disallows reads from crossing it, and `mb()` results in a read-write barrier.

In addition to the CPU-to-hardware interactions referred to previously, memory barriers are also relevant for CPU-to-CPU interactions on SMP systems. If your CPU's data cache is operating in write-back mode (in which data is not copied from cache to memory until it's absolutely necessary), you might want to stall the instruction stream until the cache-to-memory queue is drained. This is relevant, for example, when you encounter instructions that acquire or release locks. Barriers are used in this scenario to obtain a consistent perception across CPUs.

We revisit memory barriers when we discuss PCI drivers in Chapter 10 and flash map drivers in Chapter 17. In the meanwhile, stop by *Documentation/memory-barriers.txt* for an explanation of different kinds of memory barriers.

Power Management

Power management is critical on devices running on battery, such as laptops and handhelds. Linux drivers need to be aware of power states and have to transition across states in response to events such as *standby, sleep*, and *low battery.* Drivers utilize power-saving features supported by the underlying hardware when they switch to modes that consume less power. For example, the storage driver spins down the disk, whereas the video driver blanks the display.

Power-aware code in device drivers is only one piece of the overall power management framework. Power management also features participation from user space daemons, utilities, configuration files, and boot firmware. Two popular power management mechanisms are APM (discussed in the section, "Protected Mode Calls" in Appendix B, "Linux and the BIOS") and *Advanced Configuration and Power Interface* (ACPI). APM is getting obsolete, and ACPI has emerged as the de facto power management strategy on Linux systems. ACPI is further discussed in Chapter 20, "More Devices and Drivers."

Looking at the Sources

The core interrupt handling code is generic and is in the *kernel/irq/* directory. The architecture-specific portions can be found in *arch/your-arch/kernel/irq.c.* The function `do_IRQ()` defined in this file is a good place to start your journey into the kernel interrupt handling mechanism.

The kernel softirq and tasklet implementations live in *kernel/softirq.c.* This file also contains additional functions that offer more fine-grained control over softirqs and tasklets. Look at *include/linux/interrupt.h* for softirq vector enumerations and prototypes required to implement your interrupt handler. For a real-life example of writing interrupt handlers and bottom halves, start from the handler that is part of *drivers/net/lib8390.c* and follow the trail into the networking stack.

The kobject implementation and related programming interfaces live in *lib/kobject.c* and *include/linux/kobject.h.* Look at *drivers/base/sys.c* for the sysfs implementation. You will find device class APIs in *drivers/base/class.c.* Dispatching hotplug uevents via netlink sockets is done by *lib/kobject_uevent.c.* You may download udev sources and documentation from www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html.

For a fuller understanding of how APM is implemented on x86 Linux, look at *arch/x86/kernel/apm_32.c, include/linux/apm_bios.h*, and *include/asm-x86/mach-default/apm.h* in the kernel tree. If you are curious to know how APM is implemented on BIOS-less architectures such as ARM, look at *include/linux/apm-emulation.h* and its users. The kernel's ACPI implementation lives in *drivers/acpi/*.

Table 4.2 contains a summary of the main data structures used in this chapter and the location of their definitions in the source tree. Table 4.3 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 4.2. Summary of Data Structures

Data Structure	Location	Description
`tasklet_struct`	*include/linux/interrupt.h*	Manages a tasklet, which is a method to implement bottom halves
`kobject`	*include/linux/kobject.h*	Encapsulates common properties of a kernel object
`kset`	*include/linux/kobject.h*	An object set to which a kobject belongs
`kobj_type`	*include/linux/kobject.h*	An object type that describes a kobject
`class`	*include/linux/device.h*	Abstracts the idea that a driver falls under a broader category
`bus device device_driver`	*include/linux/device.h*	Structures that form the pillars under the Linux device model

Table 4.3. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`request_irq()`	*kernel/irq/manage.c*	Requests an IRQ and associates an interrupt handler with it
`free_irq()`	*kernel/irq/manage.c*	Frees an IRQ

Kernel Interface	Location	Description
`disable_irq()`	*kernel/irq/manage.c*	Disables the interrupt associated with a supplied IRQ
`disable_irq_nosync()`	*kernel/irq/manage.c*	Disables the interrupt associated with a supplied IRQ without waiting for any currently executing instances of the interrupt handler to return
`enable_irq()`	*kernel/irq/manage.c*	Re-enables the interrupt that has been disabled using `disable_irq()` or `disable_irq_nosync()`
`open_softirq()`	*kernel/softirq.c*	Opens a softirq
`raise_softirq()`	*kernel/softirq.c*	Marks the softirq as pending execution
`tasklet_init()`	*kernel/softirq.c*	Dynamically initializes a tasklet
`tasklet_schedule()`	*include/linux/interrupt.hkernel/softirq.c*	Marks a tasklet as pending execution
`tasklet_enable()`	*include/linux/interrupt.h*	Enables a tasklet
`tasklet_disable()`	*include/linux/interrupt.h*	Disables a tasklet
`tasklet_disable_nosync()`	*include/linux/interrupt.h*	Disables a tasklet without waiting for active instances to finish execution

`class_device_register()`
`kobject_add()`
`sysfs_create_dir()`
`class_device_create()`
`class_device_destroy()`
`class_create()`
`class_destroy()`
`class_device_create_file()`
`sysfs_create_file()`
`class_device_add_attrs()`
`kobject_uevent()`
	drivers/base/class.c
	lib/kobject.c
	lib/kobject_uevent.c
	fs/sysfs/dir.c
	fs/sysfs/file.c
		Family of functions in the Linux device model that create/destroy a class, device class, and associated kobjects and sysfs files

This finishes our exploration of device driver concepts. You might want to dip back into this chapter while developing your driver.

Chapter 5. Character Drivers

In This Chapter

- Char Driver Basics 120
- Device Example: System CMOS 121
- Sensing Data Availability 139
- Talking to the Parallel Port 145
- RTC Subsystem 156
- Pseudo Char Drivers 157
- Misc Drivers 160
- Character Caveats 166
- Looking at the Sources 167

You are now all set to make a foray into writing simple, albeit real-world, device drivers. In this chapter, let's look at the internals of a character (or char) device driver, which is kernel code that sequentially accesses data from a device. Char drivers can capture raw data from several types of devices: printers, mice, watchdogs, tapes, memory, RTCs, and so on. They are however, not suitable for managing data residing on block devices capable of random access such as hard disks, floppies, or compact discs.

Char Driver Basics

Let's start with a top-down view. To access a char device, a system user invokes a suitable application program. The application is responsible for talking to the device, but to do that, it needs to elicit the identity of a suitable driver. The contact details of the driver are exported to user space via the */dev* directory:

```
bash> ls -l /dev
total 0
crw-------   1 root root     5,   1 Jul 16 10:02 console
...
lrwxrwxrwx   1 root root          3 Oct 6 10:02  cdrom -> hdc
...
brw-rw----   1 root disk     3,   0 Oct 6 2007   hda
brw-rw----   1 root disk     3,   1 Oct 6 2007   hda1
...
crw-------   1 root tty      4,   1 Oct 6 10:20  tty1
crw-------   1 root tty      4,   2 Oct 6 10:02  tty2
```

The first character in each line of the `ls` output denotes the driver type: `c` signifies a char driver, `b` stands for a block driver, and `l` denotes a symbolic link. The numbers in the fifth column are called *major* numbers, and those in the sixth column are *minor* numbers. A major number broadly identifies the driver, whereas a minor number pinpoints the exact device serviced by the driver. For example, the IDE block storage driver */dev/hda* owns a major number of 3 and is in charge of handling the hard disk on your system, but when you further specify a minor number of 1 (*/dev/hda1*), that narrows it down to the first disk partition. Char and block drivers occupy different spaces, so you can have same major number assigned to a char as well as a block driver.

Let's take a step further and peek inside a char driver. From a code-flow perspective, char drivers have the following:

- An initialization (or `init()`) routine that is responsible for initializing the device and seamlessly tying the driver to the rest of the kernel via registration functions.
- A set of entry points (or methods) such as `open()`, `read()`, `ioctl()`, `llseek()`, and `write()`, which directly correspond to I/O system calls invoked by user applications over the associated */dev* node.
- Interrupt routines, bottom halves, timer handlers, helper kernel threads, and other support infrastructure. These are largely transparent to user applications.

From a data-flow perspective, char drivers own the following key data structures:

1. A per-device structure. This is the information repository around which the driver revolves.
2. `struct cdev`, a kernel abstraction for character drivers. This structure is usually embedded inside the per-device structure referred previously.
3. `struct file_operations`, which contains the addresses of all driver entry points.
4. `struct file`, which contains information about the associated */dev* node.

Chapter 5. Character Drivers

In This Chapter

- Char Driver Basics 120
- Device Example: System CMOS 121
- Sensing Data Availability 139
- Talking to the Parallel Port 145
- RTC Subsystem 156
- Pseudo Char Drivers 157
- Misc Drivers 160
- Character Caveats 166
- Looking at the Sources 167

You are now all set to make a foray into writing simple, albeit real-world, device drivers. In this chapter, let's look at the internals of a character (or char) device driver, which is kernel code that sequentially accesses data from a device. Char drivers can capture raw data from several types of devices: printers, mice, watchdogs, tapes, memory, RTCs, and so on. They are however, not suitable for managing data residing on block devices capable of random access such as hard disks, floppies, or compact discs.

Char Driver Basics

Let's start with a top-down view. To access a char device, a system user invokes a suitable application program. The application is responsible for talking to the device, but to do that, it needs to elicit the identity of a suitable driver. The contact details of the driver are exported to user space via the */dev* directory:

```
bash> ls -l /dev
total 0
crw-------   1 root root     5,   1 Jul 16 10:02 console
...
lrwxrwxrwx   1 root root          3 Oct 6 10:02  cdrom -> hdc
...
brw-rw----   1 root disk     3,   0 Oct 6 2007   hda
brw-rw----   1 root disk     3,   1 Oct 6 2007   hda1
...
crw-------   1 root tty      4,   1 Oct 6 10:20  tty1
crw-------   1 root tty      4,   2 Oct 6 10:02  tty2
```

The first character in each line of the `ls` output denotes the driver type: `c` signifies a char driver, `b` stands for a block driver, and `l` denotes a symbolic link. The numbers in the fifth column are called *major* numbers, and those in the sixth column are *minor* numbers. A major number broadly identifies the driver, whereas a minor number pinpoints the exact device serviced by the driver. For example, the IDE block storage driver */dev/hda* owns a major number of 3 and is in charge of handling the hard disk on your system, but when you further specify a minor number of 1 (*/dev/hda1*), that narrows it down to the first disk partition. Char and block drivers occupy different spaces, so you can have same major number assigned to a char as well as a block driver.

Let's take a step further and peek inside a char driver. From a code-flow perspective, char drivers have the following:

- An initialization (or `init()`) routine that is responsible for initializing the device and seamlessly tying the driver to the rest of the kernel via registration functions.
- A set of entry points (or methods) such as `open()`, `read()`, `ioctl()`, `llseek()`, and `write()`, which directly correspond to I/O system calls invoked by user applications over the associated */dev* node.
- Interrupt routines, bottom halves, timer handlers, helper kernel threads, and other support infrastructure. These are largely transparent to user applications.

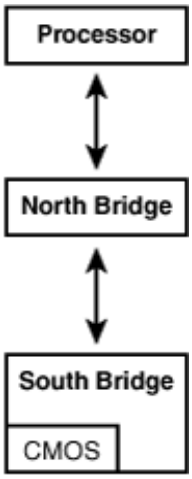
From a data-flow perspective, char drivers own the following key data structures:

1. A per-device structure. This is the information repository around which the driver revolves.
2. `struct cdev`, a kernel abstraction for character drivers. This structure is usually embedded inside the per-device structure referred previously.
3. `struct file_operations`, which contains the addresses of all driver entry points.
4. `struct file`, which contains information about the associated */dev* node.

Device Example: System CMOS

Let's implement a char driver to access the system CMOS. The BIOS on PC-compatible hardware (see Figure 5.1) uses the CMOS to store information such as startup options, boot order, and the system date, which you can configure via the BIOS setup menu. Our example CMOS driver lets you access the two PC CMOS banks as though they are regular files. Applications can operate on */dev/cmos/0* and */dev/cmos/1*, and use I/O system calls to access data from the two banks. Because the BIOS assigns semantics to the CMOS area at bit-level granularity, the driver is capable of bit-level access. So, a `read()` obtains the specified number of bits and advances the internal file pointer by the number of bits read.

Figure 5.1. CMOS on a PC-compatible system.



The CMOS is accessed via two I/O addresses, an *index* register and a *data* register, as shown in Table 5.1. You have to specify the desired CMOS memory offset in the index register and exchange information via the data register.

Table 5.1. Register Layout on the CMOS

Register Name	Description
`CMOS_BANK0_INDEX_PORT`	Specify the desired CMOS bank 0 offset in this register.
`CMOS_BANK0_DATA_PORT`	Read/write data from/to the address specified in `CMOS_BANK0_INDEX_PORT`.
`CMOS_BANK1_INDEX_PORT`	Specify the desired CMOS bank 1 offset in this register.
`CMOS_BANK1_DATA_PORT`	Read/write data from/to the address specified in `CMOS_BANK1_INDEX_PORT`.

Because each driver method has a system call counterpart that applications use, we will look at the system calls and the matching driver methods in tandem.

Driver Initialization

The driver `init()` method is the bedrock of the registration mechanism. It's responsible for the following:

- Requesting allocation of device major numbers.
- Allocating memory for the per-device structure.
- Connecting the entry points (`open()`, `read()`, and so on) with the char driver's `cdev` abstraction.
- Associating the device major number with the driver's `cdev`.
- Creating nodes under */dev* and */sys*. As discussed in Chapter 4, "Laying the Groundwork," */dev* management has meandered from static device nodes in the 2.2 kernels, to dynamic names in 2.4, and further to a user-space policy daemon (udevd) in 2.6.
- Initializing the hardware. This is not relevant for our simple CMOS.

Listing 5.1 implements the CMOS driver's `init()` method.

Listing 5.1. CMOS Driver Initialization

Code View:

```
#include <linux/fs.h>

/* Per-device (per-bank) structure */
struct cmos_dev {
  unsigned short current_pointer; /* Current pointer within the
                                     bank */
  unsigned int size;              /* Size of the bank */
  int bank_number;                /* CMOS bank number */
  struct cdev cdev;               /* The cdev structure */
  char name[10];                  /* Name of I/O region */
  /* ... */                       /* Mutexes, spinlocks, wait
                                     queues, .. */
} *cmos_devp;

/* File operations structure. Defined in linux/fs.h */
static struct file_operations cmos_fops = {
  .owner    =   THIS_MODULE,      /* Owner */
  .open     =   cmos_open,        /* Open method */
  .release  =   cmos_release,     /* Release method */
  .read     =   cmos_read,        /* Read method */
  .write    =   cmos_write,       /* Write method */
  .llseek   =   cmos_llseek,      /* Seek method */
  .ioctl    =   cmos_ioctl,       /* Ioctl method */
};

static dev_t cmos_dev_number;   /* Allotted device number */
struct class *cmos_class;       /* Tie with the device model */
```

```
#define NUM_CMOS_BANKS          2
#define CMOS_BANK_SIZE          (0xFF*8)
#define DEVICE_NAME             "cmos"
#define CMOS_BANK0_INDEX_PORT   0x70
#define CMOS_BANK0_DATA_PORT    0x71
#define CMOS_BANK1_INDEX_PORT   0x72
#define CMOS_BANK1_DATA_PORT    0x73

unsigned char addrports[NUM_CMOS_BANKS] = {CMOS_BANK0_INDEX_PORT,
                                           CMOS_BANK1_INDEX_PORT,};

unsigned char dataports[NUM_CMOS_BANKS] = {CMOS_BANK0_DATA_PORT,
                                           CMOS_BANK1_DATA_PORT,};

/*
 * Driver Initialization
 */
int __init
cmos_init(void)
{
  int i;

  /* Request dynamic allocation of a device major number */
  if (alloc_chrdev_region(&cmos_dev_number, 0,
                          NUM_CMOS_BANKS, DEVICE_NAME) < 0) {
    printk(KERN_DEBUG "Can't register device\n"); return -1;
  }

  /* Populate sysfs entries */
  cmos_class = class_create(THIS_MODULE, DEVICE_NAME);

  for (i=0; i<NUM_CMOS_BANKS; i++) {
    /* Allocate memory for the per-device structure */
    cmos_devp = kmalloc(sizeof(struct cmos_dev), GFP_KERNEL);
    if (!cmos_devp) {
      printk("Bad Kmalloc\n"); return 1;
    }

    /* Request I/O region */
    sprintf(cmos_devp->name, "cmos%d", i);
    if (!(request_region(addrports[i], 2, cmos_devp->name)) {
      printk("cmos: I/O port 0x%x is not free.\n", addrports[i]);
      return –EIO;
    }
    /* Fill in the bank number to correlate this device
       with the corresponding CMOS bank */
    cmos_devp->bank_number = i;

    /* Connect the file operations with the cdev */
    cdev_init(&cmos_devp->cdev, &cmos_fops);
    cmos_devp->cdev.owner = THIS_MODULE;

    /* Connect the major/minor number to the cdev */
    if (cdev_add(&cmos_devp->cdev, (dev_number + i), 1)) {
      printk("Bad cdev\n");
      return 1;
    }

    /* Send uevents to udev, so it'll create /dev nodes */
```

```
    class_device_create(cmos_class, NULL, (dev_number + i),
                        NULL, "cmos%d", i);
  }

  printk("CMOS Driver Initialized.\n");
  return 0;
}


/* Driver Exit */
void __exit
cmos_cleanup(void)
{
  int i;

  /* Remove the cdev */
  cdev_del(&cmos_devp->cdev);

  /* Release the major number */
  unregister_chrdev_region(MAJOR(dev_number), NUM_CMOS_BANKS);

  /* Release I/O region */
  for (i=0; i<NUM_CMOS_BANKS; i++) {
    class_device_destroy(cmos_class, MKDEV(MAJOR(dev_number), i));
    release_region(addrports[i], 2);
  }
  /* Destroy cmos_class */
  class_destroy(cmos_class);
  return();
}

module_init(cmos_init);
module_exit(cmos_cleanup);
```

Most steps performed by `cmos_init()` are generic, so if you remove references to CMOS data structures, you may use Listing 5.1 as a template to develop other char drivers, too.

First, `cmos_init()` invokes `alloc_chrdev_region()` to dynamically request an unused major number. `dev_number` contains the allotted major number if the call is successful. The second and third arguments to `alloc_chrdev_region()` specify the start minor number and the number of supported minor devices, respectively. The last argument is the device name used to identify the CMOS in */proc/devices*:

```
bash> cat /proc/devices | grep cmos
253 cmos
```

253 is the dynamically allocated major number for the CMOS device. During pre-2.6 days, dynamic device node allocation was not supported, so char drivers made calls to `register_chrdev()` to statically request specific major numbers.

Before proceeding further down the code path, let's take a peek at the data structures used in Listing 5.1. `cmos_dev` is the per-device data structure referred to earlier. `cmos_fops` is the `file_operations` structure that contains the address of driver entry points. `cmos_fops` also has a field called `owner` that is set to `THIS_MODULE`,

the address of the driver module in question. Knowing the identity of the structure owner enables the kernel to offload from the driver the burden of some housekeeping functions such as tracking the use-count when processes open or release the device.

As you saw, the kernel uses an abstraction called `cdev` to internally represent char devices. Char drivers usually embed their `cdev` inside their per-device structure. In our example, `cdev` sits inside `cmos_dev`. `cmos_init()` loops over each supported minor device (CMOS bank in this case) allocating memory for the associated per-device structure and, hence, for the `cdev` structure living inside it. `cdev_init()` associates the file operations (`cmos_fops`) with the `cdev`, and `cdev_add()` connects the major/minor numbers allocated by `alloc_chrdev_region()` to the `cdev`.

`class_create()` populates a sysfs entry for this device, and `class_device_create()` results in the generation of two uevents: cmos0 and cmos1. As you learned in Chapter 4, udevd listens to uevents and generates device nodes after consulting its rules database. Add the following to the udev rules directory (*/etc/udev/rules.d/*) to produce device nodes corresponding to the two CMOS banks (*/dev/cmos/0* and */dev/cmos/1*) on receiving the respective uevents (cmos0 and cmos1):

```
KERNEL="cmos[0-1]*", NAME="cmos/%n"
```

Device drivers that need to operate on a range of I/O addresses stake claim to the addresses via a call to `request_region()`. This regulatory mechanism ensures that requests by others for the same region fail until the occupant releases it via a call to `release_region()`. `request_region()` is commonly invoked by I/O bus drivers such as PCI and ISA to mark ownership of on-card memory in the processor's address space (more on this in Chapter 10, "Peripheral Component Interconnect"). `cmos_init()` requests access to the I/O region of each CMOS bank by calling `request_region()`. The last argument to `request_region()` is an identifier used by */proc/ioports*, so you will see this if you peek at that file:

```
bash>  grep cmos /proc/ioports
0070-0071  :  cmos0
0072-0073  :  cmos1
```

This completes the registration process, and `cmos_init()` prints out a message signaling its happiness.

Open and Release

The kernel invokes the driver's `open()` method when an application opens the corresponding device node. You can trigger execution of `cmos_open()` by doing this:

```
bash> cat /dev/cmos/0
```

The kernel calls the `release()` method when an application closes an open device. So when *cat* closes the file descriptor attached to */dev/cmos/0* after reading the contents of CMOS bank 0, the kernel invokes `cmos_release()`.

Listing 5.2 shows the implementation of `cmos_open()` and `cmos_release()`. Let's take a closer look at `cmos_open()`. There are a couple of things worthy of note here. The first is the extraction of `cmos_dev`. The inode passed as an argument to `cmos_open()` contains the address of the `cdev` structure allocated during initialization. As shown in Listing 5.1, `cdev` is embedded inside `cmos_dev`. To elicit the address of the container structure `cmos_dev`, `cmos_open()` uses the kernel helper function, `container_of()`.

The other notable operation in `cmos_open()` is the usage of the `private_data` field that is part of `struct file`, the second argument. You can use this field (`file->private_data`) as a placeholder to conveniently correlate information from inside other driver methods. The CMOS driver uses this field to store the address of `cmos_dev`.

Look at `cmos_release()` (and the rest of the methods) to see how `private_data` is used to directly obtain a handle on the `cmos_dev` structure belonging to the corresponding CMOS bank.

Listing 5.2. Open and Release

Code View:

```
/*
 * Open CMOS bank
 */
int
cmos_open(struct inode *inode, struct file *file)
{
  struct cmos_dev *cmos_devp;

  /* Get the per-device structure that contains this cdev */
  cmos_devp = container_of(inode->i_cdev, struct cmos_dev, cdev);

  /* Easy access to cmos_devp from rest of the entry points */
  file->private_data = cmos_devp;

  /* Initialize some fields */
  cmos_devp->size = CMOS_BANK_SIZE;
  cmos_devp->current_pointer = 0;

  return 0;
}

/*
 * Release CMOS bank
 */
int
cmos_release(struct inode *inode, struct file *file)
{
  struct cmos_dev *cmos_devp = file->private_data;

  /* Reset file pointer */
  cmos_devp->current_pointer = 0;

  return 0;
}
```

Exchanging Data

`read()` and `write()` are the basic char driver methods responsible for exchanging data between user space and the device. The extended `read()`/`write()` family contains several other methods, too: `fsync()`, `aio_read()`, `aio_write()`, and `mmap()`.

The CMOS driver operates on a simple memory device and does not have to work through some of the complexities faced by usual char drivers:

- CMOS data access routines do not need to sleep-wait for device I/O to complete, whereas `read()` and `write()` methods belonging to many char drivers have to support both blocking and nonblocking modes of operation. Unless a device file is opened in the nonblocking (`O_NONBLOCK`) mode, `read()` and `write()` are allowed to put the calling process to sleep until the corresponding operation completes.

- CMOS driver operations complete synchronously and do not depend on interrupts. However, data access methods belonging to many drivers depend on interrupts for data collection and have to communicate with interrupt context code via data structures such as wait queues.

Listing 5.3 contains the `read()` and `write()` methods belonging to the CMOS driver. You cannot directly access user buffers from kernel space and vice versa, so to copy CMOS memory contents to user space, `cmos_read()` uses the services of `copy_to_user()`. `cmos_write()` does the reverse using `copy_from_user()`. Because `copy_to_user()` and `copy_from_user()` may fall asleep on the job, you cannot hold spinlocks while calling them.

As you saw earlier, accessing CMOS memory is accomplished by operating on a pair of I/O addresses. To read different sizes of data from an I/O address, the kernel provides a family of architecture-independent functions: `in[b|w|l|sb|sl]()`. Similarly, a cluster of routines, `out[b|w|l|sb|sl]()`, are available for writing to I/O regions. `port_data_in()` and `port_data_out()` in Listing 5.3 use `inb()` and `oub()` for data transfer.

Listing 5.3. Read and Write

Code View:

```
/*
 * Read from a CMOS Bank at bit-level granularity
 */
ssize_t
cmos_read(struct file *file, char *buf,
          size_t count, loff_t *ppos)
{
  struct cmos_dev *cmos_devp = file->private_data;
  char data[CMOS_BANK_SIZE];
  unsigned char mask;
  int xferred = 0, i = 0, l, zero_out;
  int start_byte = cmos_devp->current_pointer/8;
  int start_bit  = cmos_devp->current_pointer%8;

  if (cmos_devp->current_pointer >= cmos_devp->size) {
    return 0; /*EOF*/
  }

  /* Adjust count if it edges past the end of the CMOS bank */
  if (cmos_devp->current_pointer + count > cmos_devp->size) {
    count = cmos_devp->size - cmos_devp->current_pointer;
  }

  /* Get the specified number of bits from the CMOS */
  while (xferred < count) {
    data[i] = port_data_in(start_byte, cmos_devp->bank_number)
              >> start_bit;
    xferred += (8 - start_bit);
    if ((start_bit) && (count + start_bit > 8)) {
      data[i] |= (port_data_in (start_byte + 1,
                  cmos_devp->bank_number) << (8 - start_bit));
      xferred += start_bit;
    }
    start_byte++;
    i++;
  }
  if (xferred > count) {
```

```
    /* Zero out (xferred-count) bits from the MSB
       of the last data byte */
    zero_out = xferred - count;
    mask = 1 << (8 - zero_out);
    for (l=0; l < zero_out; l++) {
      data[i-1] &= ~mask; mask <<= 1;
    }
    xferred = count;
  }

  if (!xferred) return -EIO;

  /* Copy the read bits to the user buffer */
  if (copy_to_user(buf, (void *)data, ((xferred/8)+1)) != 0) {
    return -EIO;
  }

  /* Increment the file pointer by the number of xferred bits */
  cmos_devp->current_pointer += xferred;
  return xferred; /* Number of bits read */
}


/*
 * Write to a CMOS bank at bit-level granularity. 'count' holds the
 * number of bits to be written.
 */
ssize_t
cmos_write(struct file *file, const char *buf,
           size_t count, loff_t *ppos)
{
  struct cmos_dev *cmos_devp = file->private_data;
  int xferred = 0, i = 0, l, end_l, start_l;
  char *kbuf, tmp_kbuf;
  unsigned char tmp_data = 0, mask;
  int start_byte = cmos_devp->current_pointer/8;
  int start_bit  = cmos_devp->current_pointer%8;

  if (cmos_devp->current_pointer >= cmos_devp->size) {
    return 0; /* EOF */
  }
  /* Adjust count if it edges past the end of the CMOS bank */
  if (cmos_devp->current_pointer + count > cmos_devp->size) {
    count = cmos_devp->size - cmos_devp->current_pointer;
  }

  kbuf = kmalloc((count/8)+1,GFP_KERNEL);
  if (kbuf==NULL)
    return -ENOMEM;

  /* Get the bits from the user buffer */
  if (copy_from_user(kbuf,buf,(count/8)+1)) {
    kfree(kbuf);
    return -EFAULT;
  }

  /* Write the specified number of bits to the CMOS bank */
  while (xferred < count) {
    tmp_data = port_data_in(start_byte, cmos_devp->bank_number);
```

```
    mask = 1 << start_bit;
    end_l = 8;
    if ((count-xferred) < (8 - start_bit)) {
      end_l = (count - xferred) + start_bit;
    }

    for (l = start_bit; l < end_l; l++) {
      tmp_data &= ~mask; mask <<= 1;
    }
    tmp_kbuf = kbuf[i];
    mask = 1 << end_l;
    for (l = end_l; l < 8; l++) {
      tmp_kbuf &= ~mask;
      mask <<= 1;
    }

    port_data_out(start_byte,
                  tmp_data |(tmp_kbuf << start_bit),
                  cmos_devp->bank_number);
    xferred += (end_l - start_bit);

    if ((xferred < count) && (start_bit) &&
        (count + start_bit > 8)) {
      tmp_data = port_data_in(start_byte+1,
                              cmos_devp->bank_number);
      start_l = ((start_bit + count) % 8);
      mask = 1 << start_l;
      for (l=0; l < start_l; l++) {
        mask >>= 1;
        tmp_data &= ~mask;
      }
      port_data_out((start_byte+1),
                    tmp_data |(kbuf[i] >> (8 - start_bit)),
                    cmos_devp->bank_number);
      xferred += start_l;
    }

    start_byte++;
    i++;
  }

  if (!xferred) return -EIO;

  /* Push the offset pointer forward */
  cmos_devp->current_pointer += xferred;
  return xferred; /* Return the number of written bits */
}


/*
 * Read data from specified CMOS bank
 */
unsigned char
port_data_in(unsigned char offset, int bank)
{
  unsigned char data;

  if (unlikely(bank >= NUM_CMOS_BANKS)) {
    printk("Unknown CMOS Bank\n");
```

```
    return 0;
  } else {
    outb(offset, addrports[bank]); /* Read a byte */
    data = inb(dataports[bank]);
  }
  return data;


}
/*
 * Write data to specified CMOS bank
 */
void
port_data_out(unsigned char offset, unsigned char data,
                int bank)
{
  if (unlikely(bank >= NUM_CMOS_BANKS)) {
    printk("Unknown CMOS Bank\n");
    return;
  } else {
    outb(offset, addrports[bank]); /* Output a byte */
    outb(data, dataports[bank]);
  }
  return;
}
```

If a char driver's `write()` method returns successfully, it implies that the driver has assumed responsibility for the data passed down to it by the application. However it does not guarantee that the data has been successfully written to the device. If an application needs this assurance, it can invoke the `fsync()` system call. The corresponding `fsync()` driver method ensures that application data is flushed from driver buffers and written to the device. The CMOS driver does not need an `fsync()` method because, in this case, driver-writes are synonymous with device-writes.

If a user application has data sitting on multiple buffers that it needs to send to a device, it can request multiple driver writes, but that is inefficient for the following reasons:

1. The overhead of multiple system calls and related context switches.

2. The driver is the one who knows the device intimately, so it can probably do a more clever job of efficiently gathering data from different buffers and dispatching it to the device.

Because of this, vectored versions of `read()` and `write()` are supported on Linux and other UNIX flavors. The Linux char driver infrastructure used to offer two dedicated methods to perform vector operations: `readv()` and `writev()`. Starting with the 2.6.19 kernel release, these two methods have been folded into the generic *Linux Asynchronous I/O* (AIO) layer, however. Linux AIO is a broad topic and is outside the scope of this discussion, so we just concentrate on the synchronous vector capabilities offered by AIO.

The prototypes of the vector driver methods are as follows:

```
ssize_t aio_read(struct kiocb *iocb, const struct iovec *vector,
                 unsigned long count, loff_t offset);
ssize_t aio_write(struct kiocb *iocb, const struct iovec *vector,
                  unsigned long count, loff_t offset);
```

The first argument to `aio_read()/aio_write()` describes the AIO operation, and the second argument is an array of `iovecs`. The latter is the principal data structure used by the vector functions and contains the addresses and lengths of buffers that hold the data. In fact, this mechanism is the user space equivalent of scatter-gather DMA discussed in Chapter 10. Look at *include/linux/uio.h* for the definition of `iovecs` and at *drivers/net/tun.c*[1] for an example implementation of vectored char driver methods.

[1] Discussed in the sidebar "TUN/TAP Driver" in Chapter 15, "Network Interface Cards."

Another data access method is `mmap()`, which associates device memory with user virtual memory. Applications may call the corresponding system call, also called `mmap()`, and directly operate on the returned memory region to access device-resident memory. Not many drivers implement `mmap()`, so we won't delve into that here. Instead, have a look at *drivers/char/mem.c* for an example `mmap()` implementation. The section "Accessing Memory Regions" in Chapter 19, "Drivers in User Space," illustrates how applications use `mmap()`. Our example CMOS driver does not implement `mmap()`.

You might have noticed that `port_data_in()` and `port_data_out()` envelop the bank number sanity check within a macro called `unlikely()`. Two macros, `likely()` and `unlikely()`, inform GCC about the probability of success of the associated conditional evaluation. This information is used by GCC while predicting branches. Because we mark it unlikely that the bank sanity check will fail, GCC generates intelligent code that gels the `else{}` clause sequentially with the code flow. Branching is done for the `if{}` clause. The reverse happens if you use `likely()` rather than `unlikely()`.

Seek

The kernel uses an internal pointer to keep track of the current file access position. Applications use the `lseek()` system call to request repositioning of this internal file pointer. Using the services of `lseek()`, you can reset the file pointer to any offset within the file. The char driver counterpart of `lseek()` is the `llseek()` method. `cmos_llseek()` implements this method in the CMOS driver.

As we saw previously, the internal file pointer for the CMOS moves bit-wise rather than byte-wise. If a byte of data is read from the CMOS driver, the file pointer has to be moved by 8, so applications have to seek accordingly. `cmos_llseek()` also implements end-of-file semantics depending on the size of the CMOS bank.

To understand the semantics of `llseek()`, let's start by looking at the commands supported by the `lseek()` system call:

1. `SEEK_SET`, which sets the file pointer to a supplied fixed offset.

2. `SEEK_CUR`, which calculates the offset relative to the current location.

3. `SEEK_END`, which calculates the offset relative to the end-of-file. This command can maneuver the file pointer beyond the end of the file, but does not change the file size. Reads beyond the end-of-file marker return naught if no data is explicitly written. This technique is often used to create big files. The CMOS driver does not support `SEEK_END`.

Look at `cmos_llseek()` in Listing 5.4 and co-relate with the preceding definitions.

Listing 5.4. Seek

Code View:

```
/*
 * Seek to a bit offset within a CMOS bank
 */
static loff_t
cmos_llseek(struct file *file, loff_t offset,
            int orig)
{
  struct cmos_dev *cmos_devp = file->private_data;

  switch (orig) {
    case 0: /* SEEK_SET */
      if (offset >= cmos_devp->size) {
        return -EINVAL;
      }
      cmos_devp->current_pointer = offset; /* Bit Offset */
      break;

    case 1: /* SEEK_CURR */
      if ((cmos_devp->current_pointer + offset) >=
           cmos_devp->size) {
        return -EINVAL;
      }
      cmos_devp->current_pointer = offset; /* Bit Offset */
      break;

    case 2: /* SEEK_END - Not supported */
      return -EINVAL;

    default:
      return -EINVAL;
  }

  return(cmos_devp->current_pointer);
}
```

Control

Another common char driver method is called I/O Control (or *ioctl*). This routine is used to receive and implement application commands that request device-specific actions. Because CMOS memory is used by the BIOS to store crucial information such as the boot device order, it's usually protected via *cyclic redundancy check* (CRC) algorithms. To detect data corruption, the CMOS driver supports two ioctl commands:

1. *Adjust checksum*, which is used to recalculate the CRC after the CMOS contents have been modified. The calculated checksum is stored at a predetermined offset in CMOS bank 1.

2. *Verify checksum*, which is used to check whether the CMOS contents are healthy. This is done by comparing the CRC of the current contents with the value previously stored.

Applications send these commands down to the driver via the `ioctl()` system call when they want to request it to perform checksum operations. Look at `cmos_ioctl()` in Listing 5.5 for the implementation of the CMOS driver's `ioctl` method. `adjust_cmos_crc(int bank, unsigned short seed)` implements the standard CRC algorithm and is not shown in the listing.

Listing 5.5. I/O Control

Code View:

```
#define CMOS_ADJUST_CHECKSUM 1
#define CMOS_VERIFY_CHECKSUM 2

#define CMOS_BANK1_CRC_OFFSET 0x1E

/*
 * Ioctls to adjust and verify CRC16s.
 */
static int
cmos_ioctl(struct inode *inode, struct file *file,
           unsigned int cmd, unsigned long arg)
{
  unsigned short crc = 0;
  unsigned char buf;

  switch (cmd) {
    case CMOS_ADJUST_CHECKSUM:
      /* Calculate the CRC of bank0 using a seed of 0 */
      crc = adjust_cmos_crc(0, 0);

      /* Seed bank1 with CRC of bank0 */
      crc = adjust_cmos_crc(1, crc);

      /* Store calculated CRC */
      port_data_out(CMOS_BANK1_CRC_OFFSET,
                    (unsigned char)(crc & 0xFF), 1);
      port_data_out((CMOS_BANK1_CRC_OFFSET + 1),
                    (unsigned char) (crc >> 8), 1);
      break;

    case CMOS_VERIFY_CHECKSUM:
     /* Calculate the CRC of bank0 using a seed of 0 */
      crc = adjust_cmos_crc(0, 0);

     /* Seed bank1 with CRC of bank0 */
     crc = adjust_cmos_crc(1, crc);

     /* Compare the calculated CRC with the stored CRC */
     buf = port_data_in(CMOS_BANK1_CRC_OFFSET, 1);
     if (buf != (unsigned char) (crc & 0xFF)) return -EINVAL;

     buf = port_data_in((CMOS_BANK1_CRC_OFFSET+1), 1);
     if (buf != (unsigned char)(crc >> 8)) return -EINVAL;
     break;
     default:
```

```
        return -EIO;
    }

    return 0;
}
```

Sensing Data Availability

Many user applications are sophisticated and are not satisfied with the vintage `open()`/`read()`/`write()`/`close()` calls. They desire synchronous or asynchronous notifications that alert them when new data is available from the device or when the driver is ready to accept new data. In this section, we examine two char driver methods that sense data availability: `poll()` and `fasync()`. The former is synchronous, whereas the latter is asynchronous. Because these mechanisms are relatively advanced, let's first understand how applications use these features before finding out how the underlying driver implements them. Sensing data availability is not relevant for the simple CMOS memory device discussed previously, so let's take a few usage scenarios from a popular user space application: the X Windows server.

Poll

Consider the following code snippet from the X Windows source tree (downloadable from www.xfree86.org) that handles mice events:

***xc/programs/Xserver/hw/xfree86/input/mouse/mouse.c*:**

```
case PROT_THINKING:           /* ThinkingMouse */
  /* This mouse may send a PnP ID string, ignore it. */
  usleep(200000); xf86FlushInput(pInfo->fd);
  /* Send the command to initialize the beast. */
  for (s = "E5E5"; *s; ++s) {
    xf86WriteSerial(pInfo->fd, s, 1);
    if ((xf86WaitForInput(pInfo->fd, 1000000) <= 0))
    break;
    xf86ReadSerial(pInfo->fd, &c, 1);
    if (c != *s) break;
  }
  break;
```

Essentially, the code sends an initialization command to the mouse, polls until it senses input data, and reads the response from the device. If you peel the envelope off `Xf86WaitForInput()` used previously, you will find a call to the `select()` system call:

Code View:

***xc/programs/Xserver/hw/xfree86/os-support/shared/posix_tty.c*:**

```
int
xf86WaitForInput(int fd, int timeout)
{
  fd_set readfds;
  struct timeval to;
  int r;

  FD_ZERO(&readfds);
  if (fd >= 0) {
    FD_SET(fd, &readfds);
  }

  to.tv_sec  = timeout / 1000000;
  to.tv_usec = timeout % 1000000;

  if (fd >= 0) {
```

```
    SYSCALL (r = select(FD_SETSIZE, &readfds, NULL, NULL, &to));
  } else {
    SYSCALL (r = select(FD_SETSIZE, NULL, NULL, NULL, &to));
  }

  if (xf86Verbose >= 9)
    ErrorF ("select returned %d\n", r);

  return (r);
}
```

You may supply a bunch of file descriptors to `select()` and ask it to keep an eye on them until there is a change in the associated data state. You may also request a timeout to override data availability. If you ask for a timeout of `NULL`, `select()` blocks forever. Refer to the man or info pages of `select()` for detailed documentation. The call to `select()` in the preceding snippet induces the X server to poll for data from a connected mouse within a timeout.

> Linux supports another system call, `poll()`, which has semantics similar to `select()`. The 2.6 kernel supports a new non-POSIX system call named `epoll()` that is a more scalable superset of `poll()`. All these system calls rely on the same underlying char driver method, `poll()`.

Most I/O system calls are POSIX-compliant and are not Linux-specific (programs such as X Windows after all, run on many UNIX flavors, not just on Linux), but the internal driver methods are operating system-specific. On Linux, the `poll()` driver method is the pillar under the `select()` system call. In the previous X server scenario, the mouse driver's `poll()` method looks like this:

```
static DECLARE_WAIT_QUEUE_HEAD(mouse_wait); /* Wait Queue */

static unsigned int
mouse_poll(struct file *file, poll_table *wait)
{
  poll_wait(file, &mouse_wait, wait);
  spin_lock_irq(&mouse_lock);

  /* See if data has arrived from the device or
     if the device is ready to accept more data */
  /* ... */
  spin_unlock_irq(&mouse_lock);

  /* Availability of data is detected from interrupt context */
  if (data_is_available()) return(POLLIN | POLLRDNORM);

  /* Data can be written. Not relevant for mice */
  if (data_can_be_written()) return(POLLOUT | POLLWRNORM);

  return 0;
}
```

When `Xf86WaitForInput()` invokes `select()`, the generic kernel poll implementation (defined in *fs/select.c*) calls `mouse_poll()`. `mouse_poll()` takes two arguments, the usual file pointer (`struct file *`) and a pointer to a kernel data structure called the `poll_table`. The `poll_table` is a table of wait queues owned by device drivers that are being polled for data.

`mouse_poll()` uses the library function, `poll_wait()`, to add a wait queue (`mouse_wait`) to the kernel `poll_table` and go to sleep. As you saw in Chapter 3, "Kernel Facilities," device drivers usually own several wait queues that block until they detect a change in a data condition. This condition can be the arrival of new data from the device, willingness of the driver to pass new data to the application, or the readiness of the device (or the driver) to accept new data. Such conditions are usually (but not always) detected by the driver's interrupt handler. When the mouse driver's interrupt handler senses mouse movement, it calls `wake_up_interruptible(&mouse_wait)` to wake up the sleeping `mouse_poll()`.

If there is no change in the data condition, the `poll()` method returns `0`. If the driver is ready to send at least one byte of data to the application, it returns `POLLIN|POLLRDNORM`. If the driver is ready to accept at least a byte of data from the application, it returns `POLLOUT|POLLWRNORM`.[2] Thus, if there is no mouse movement, `mouse_poll()` returns `0`, and the calling thread is put to sleep. The kernel invokes `mouse_poll()` again when the mouse interrupt handler senses device data and wakes up the `mouse_wait` queue. This time around, `mouse_poll()` returns `POLLIN|POLLRDNORM`, so the `select()` call and hence `Xf86WaitForInput()` return positive values. The X server's mouse handler (*xc/programs/Xserver/hw/xfree86/input/mouse/mouse.c*) goes on to read data from the mouse.

[2] The full list of return codes is defined in *include/asm-generic/poll.h*. Some of them are used only by the networking stack.

User applications that poll a driver are usually more interested in driver characteristics than device characteristics. For example, depending on the health of its buffers, a driver might be ready to accept new data from the application before the device itself is.

Fasync

Some applications, for performance reasons, desire asynchronous notifications from the device driver. Assume that an application on a Linux pacemaker programmer device is busy performing complex computations but wants to be notified as soon as data arrives from an implanted pacemaker via a telemetry interface. The `select()/poll()` mechanism is not of use in this case because it blocks the computations. What the application needs is an asynchronous event report. If the telemetry driver can asynchronously dispatch a signal (usually `SIGIO`) as soon as it detects data from the pacemaker, the application can catch it using a signal handler and accordingly steer the code flow.

For a real-world example of asynchronous notification, let's revert to a region of the X server that requests alerts when data is detected from input devices. Take a look at this snippet from the X server sources:

Code View:
***xc/programs/Xserver/hw/xfree86/os-support/shared/sigio.c*:**
```
int xf86InstallSIGIOHandler(int fd, void (*f)(int, void *),
                            void *closure)
{
  struct sigaction sa;
  struct sigaction osa;
```

```
  if (fcntl(fd, F_SETOWN, getpid()) == -1) {
    blocked = xf86BlockSIGIO();

  /* O_ASYNC is defined as SIGIO elsewhere by the X server */
  if (fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_ASYNC) == -1) {
    xf86UnblockSIGIO(blocked); return 0;
  }
  sigemptyset(&sa.sa_mask);
  sigaddset(&sa.sa_mask, SIGIO);
  sa.sa_flags = 0;
  sa.sa_handler = xf86SIGIO;
  sigaction(SIGIO, &sa, &osa);
  /* ... */
  return 0;
}

static void
xf86SIGIO(int sig)
{
  /* Identify the device that triggered generation of this
     SIGIO and handle the data arriving from it */
  /* ... */
}
```

As you can decipher from the above snippet, the X server does the following:

- Calls `fcntl(F_SETOWN)`. The `fcntl()` system call is used to manipulate file descriptor behavior. `F_SETOWN` sets the ownership of the descriptor to the calling process. This is required since the kernel needs to know where to send the asynchronous signal. This step is transparent to the device driver.

- Invokes `fcntl(F_SETFL)`. `F_SETFL` requests the driver to deliver `SIGIO` to the application whenever there is data to be read, or if the driver is ready to receive more application data. The invocation of `fcntl(F_SETFL)` results in the invocation of the `fasync()` driver method. It's this method's responsibility to add or remove entries from the list of processes that are to be delivered `SIGIO`. To this end, `fasync()` utilizes the services of a kernel library function called `fasync_helper()`.

- Implements the `SIGIO` signal handler, `xf86SIGIO()`, as per its code architecture and installs it using the `sigaction()` system call. When the underlying input device driver detects a change in data status, it dispatches `SIGIO` to registered requesters and this triggers execution of `xf86SIGIO()`.[3] Char drivers call `kill_fasync()` to send `SIGIO` to registered processes. To notify a read event, `POLLIN` is passed as the argument to `kill_fasync()`. To notify a write event, the argument is `POLLOUT`.

[3] If your signal handler services asynchronous events from multiple devices, you will need additional mechanisms, such as a `select()` call inside the handler, to figure out the identity of the device responsible for the event.

To see how the driver-side of the asynchronous notification chain is implemented, let's look at a fictitious

`fasync()` method belonging to the driver of an input device:

Code View:
```
/* This is invoked by the kernel when the X server opens this
 * input device and issues fcntl(F_SETFL) on the associated file
 * descriptor. fasync_helper() ensures that if the driver issues a
 * kill_fasync(), a SIGIO is dispatched to the owning application.
 */
static int
inputdevice_fasync(int fd, struct file *filp, int on)
{
  return fasync_helper(fd, filp, on, &inputdevice_async_queue);
}
/* Interrupt Handler */
irqreturn_t
inputdevice_interrupt(int irq, void *dev_id)
{
  /* ... */
  /* Dispatch a SIGIO using kill_fasync() when input data is
     detected. Output data is not relevant since this is a read-only
     device */
  wake_up_interruptible(&inputdevice_wait);
  kill_fasync(&inputdevice_async_queue, SIGIO, POLL_IN);
  /* ... */
  return IRQ_HANDLED;
}
```

To see how `SIGIO` delivery can be complex, consider the case of a tty driver (discussed in Chapter 6, "Serial Drivers"). Interested applications get notified under different scenarios:

- If the underlying driver is not ready to accept application data, it puts the calling process to sleep. When the driver interrupt handler subsequently decides that the device can accept more data, it wakes the application and invokes `kill_fasync(POLLOUT)`.
- If a newline character is received, the tty layer calls `kill_fasync(POLLIN)`.
- When the driver wakes up a sleeping reader thread after detecting that sufficient data bytes beyond a threshold have arrived from a device, it sends that information to stakeholder processes by invoking `kill_fasync(POLLIN)`.

Talking to the Parallel Port

The parallel port is a ubiquitous 25-pin interface popularly found on PC-compatible systems. The capability of a parallel port (whether it's unidirectional, bidirectional, supports DMA, and so on) depends on the underlying chipset. Look at Figure 4.1 in Chapter 4 to find out how the PC architecture supports parallel ports.

The *drivers/parport/* directory contains code (called parport) that implements IEEE 1284 parallel port communication. Several devices that connect to the parallel port such as printers and scanners use parport's services. Parport has an architecture-independent module called *parport.ko* and an architecture-dependent one (*parport_pc.ko* for the PC architecture) that provide programming interfaces to drivers of devices that interface via the parallel port.

Let's take the example of the parallel printer driver, *drivers/char/lp.c.* These are the high-level steps needed to print a file:

1. The printer driver creates char device nodes */dev/lp0* to */dev/lpN*, one per connected printer.

2. The *Common UNIX Printing System* (CUPS) is the framework that provides print capabilities on Linux. The CUPS configuration file (*/etc/printers.conf* on some distributions) maps printers with their char device nodes (*/dev/lpX*).

3. CUPS utilities consult this file and stream data to the corresponding device node. So, if you have a printer connected to the first parallel port on your system and you issue the command, *lpr myfile*, it's streamed via */dev/lp0* to the printer's `write()` method, `lp_write()`, defined in *drivers/char/lp.c*.

4. `lp_write()` uses the services of parport to send the data to the printer.

Apple Inc. has acquired ownership of CUPS software. The code continues to be licensed under GPLv2.

A char driver called *ppdev*(*drivers/char/ppdev.c*) exports the */dev/parportX* device nodes that let user applications directly communicate with the parallel port. (We talk more about ppdev in Chapter 19.)

Device Example: Parallel Port LED Board

To learn how to use the services offered by parport, let's write a simple driver. Consider a board that has eight light-emitting diodes (LEDs) interfaced to a standard 25-pin parallel port connector. Because the 8-bit parallel port data register on the PC is directly mapped to pins 2 to 9 of the parallel port connector, those pins are wired to the LEDs on the board. Writing to the parallel port data register controls the voltage levels of these pins and turns the LEDs on or off. Listing 5.6 implements a char driver that communicates with this board over the system parallel port. Embedded comments explain the parport service routines that Listing 5.6 uses.

Listing 5.6. Driver for the Parallel LED Board (led.c)

Code View:

```
#include <linux/fs.h>
#include <linux/cdev.h>
```

```
#include <linux/parport.h>
#include <asm/uaccess.h>
#include <linux/platform_device.h>

#define DEVICE_NAME    "led"

static dev_t dev_number;          /* Allotted device number */
static struct class *led_class;   /* Class to which this device
                                     belongs */
struct cdev led_cdev;             /* Associated cdev */
struct pardevice *pdev;           /* Parallel port device */

/* LED open */
int
led_open(struct inode *inode, struct file *file)
{
  return 0;
}

/* Write to the LED */
ssize_t
led_write(struct file *file, const char *buf,
          size_t count, loff_t *ppos)
{
  char kbuf;

  if (copy_from_user(&kbuf, buf, 1)) return -EFAULT;

  /* Claim the port */
  parport_claim_or_block(pdev);

  /* Write to the device */
  parport_write_data(pdev->port, kbuf);

  /* Release the port */
  parport_release(pdev);

  return count;
}
/* Release the device */
int
led_release(struct inode *inode, struct file *file)
{
  return 0;
}

/* File Operations */
static struct file_operations led_fops = {
  .owner = THIS_MODULE,
  .open = led_open,
  .write = led_write,
  .release = led_release,
};

static int
led_preempt(void *handle)
{
  return 1;
}
```

```
/* Parport attach method */
static void
led_attach(struct parport *port)
{
  /* Register the parallel LED device with parport */
  pdev = parport_register_device(port, DEVICE_NAME,
                                 led_preempt, NULL,
                                 NULL, 0, NULL);
  if (pdev == NULL) printk("Bad register\n");
}

/* Parport detach method */
static void
led_detach(struct parport *port)
{
  /* Do nothing */
}

/* Parport driver operations */
static struct parport_driver led_driver = {
  .name   = "led",
  .attach = led_attach,
  .detach = led_detach,
};
/* Driver Initialization */
int __init
led_init(void)
{
  /* Request dynamic allocation of a device major number */
  if (alloc_chrdev_region(&dev_number, 0, 1, DEVICE_NAME)
                          < 0) {
    printk(KERN_DEBUG "Can't register device\n");
    return -1;
  }

  /* Create the led class */
  led_class = class_create(THIS_MODULE, DEVICE_NAME);
  if (IS_ERR(led_class)) printk("Bad class create\n");

  /* Connect the file operations with the cdev */
  cdev_init(&led_cdev, &led_fops);

  led_cdev.owner = THIS_MODULE;

  /* Connect the major/minor number to the cdev */
  if (cdev_add(&led_cdev, dev_number, 1)) {
    printk("Bad cdev add\n");
    return 1;
  }

  class_device_create(led_class, NULL, dev_number,
                                 NULL, DEVICE_NAME);

  /* Register this driver with parport */
  if (parport_register_driver(&led_driver)) {
    printk(KERN_ERR "Bad Parport Register\n");
    return -EIO;
  }
```

```
  printk("LED Driver Initialized.\n");
  return 0;
}

/* Driver Exit */
void __exit
led_cleanup(void)
{
  unregister_chrdev_region(MAJOR(dev_number), 1);
  class_device_destroy(led_class, MKDEV(MAJOR(dev_number), 0));
  class_destroy(led_class);
  return;
}

module_init(led_init);
module_exit(led_cleanup);

MODULE_LICENSE("GPL");
```

`led_init()` is similar to `cmos_init()` developed in Listing 5.1, but for a couple of things:

1. As you saw in Chapter 4, the new device model distinguishes between drivers and devices. `led_init()` registers the LED driver with parport via a call to `parport_register_driver()`.When the kernel finds the LED board during `led_attach()`, it registers the device by invoking `parport_register_device()`.

2. `led_init()` creates the device node */dev/led*, which you can use to control the state of individual LEDs.

Compile and insert the driver module into the kernel:

```
bash> make –C /path/to/kerneltree/ M=$PWD modules
bash> insmod ./led.ko
LED Driver Initialized
```

To selectively drive some parallel port pins and glow the corresponding LEDs, echo the appropriate value to */dev/led*:

```
bash> echo 1 > /dev/led
```

Because ASCII for `1` is `31` (or `00110001`), the first, fifth, and sixth LEDs should turn on.

The preceding command triggers invocation of `led_write()`. This driver method first copies user memory (the value 31 in this case) to kernel buffers via `copy_from_user()`. It then claims the parallel port, writes data, and releases the port, all using parport interfaces.

Sysfs is a better place than */dev* to control device state, so it's a good idea to entrust LED control to sysfs files. Listing 5.7 contains the driver implementation that achieves this. The sysfs manipulation code in the listing can

serve as a template to achieve device control from other drivers, too.

Listing 5.7. Using Sysfs to Control the Parallel LED Board

Code View:

```
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/parport.h>
#include <asm/uaccess.h>
#include <linux/pci.h>

static dev_t dev_number;         /* Allotted Device Number */
static struct class *led_class;  /* Class Device Model */
struct cdev led_cdev;            /* Character dev struct */
struct pardevice *pdev;          /* Parallel Port device */

struct kobject kobj;             /* Sysfs directory object */

/* Sysfs attribute of the leds */
struct led_attr {
  struct attribute attr;
  ssize_t (*show)(char *);
  ssize_t (*store)(const char *, size_t count);
};

#define glow_show_led(number)                           \
static ssize_t                                          \
glow_led_##number(const char *buffer, size_t count)     \
{                                                       \
  unsigned char buf;                                    \
  int value;                                            \
                                                        \
  sscanf(buffer, "%d", &value);                         \
                                                        \
  parport_claim_or_block(pdev);                         \
  buf = parport_read_data(pdev->port);                  \
  if (value) {                                          \
    parport_write_data(pdev->port, buf | (1<<number));  \
  } else {                                              \
    parport_write_data(pdev->port, buf & ~(1<<number)); \
  }                                                     \
  parport_release(pdev);                                \
  return count;                                         \
}                                                       \
                                                        \
static ssize_t                                          \
show_led_##number(char *buffer)                         \
{                                                       \
  unsigned char buf;                                    \
                                                        \
  parport_claim_or_block(pdev);                         \
                                                        \
  buf = parport_read_data(pdev->port);                  \
  parport_release(pdev);                                \
                                                        \
  if (buf & (1 << number)) {                            \
    return sprintf(buffer, "ON\n");                     \
  } else {                                              \
    return sprintf(buffer, "OFF\n");                    \
```

```
  }                                                     \
}                                                       \
                                                        \
static struct led_attr led##number =                    \
__ATTR(led##number, 0644, show_led_##number, glow_led_##number);

glow_show_led(0); glow_show_led(1); glow_show_led(2);
glow_show_led(3); glow_show_led(4); glow_show_led(5);
glow_show_led(6); glow_show_led(7);

#define DEVICE_NAME "led"

static int
led_preempt(void *handle)
{
  return 1;
}

/* Parport attach method */
static void
led_attach(struct parport *port)
{
  pdev = parport_register_device(port, DEVICE_NAME,
                                 led_preempt, NULL, NULL, 0,
                                 NULL);
  if (pdev == NULL) printk("Bad register\n");
}
/* Parent sysfs show() method. Calls the show() method
   corresponding to the individual sysfs file */
static ssize_t
l_show(struct kobject *kobj, struct attribute *a, char *buf)
{
  int ret;
  struct led_attr *lattr = container_of(a, struct led_attr,attr);

  ret = lattr->show ? lattr->show(buf) : -EIO;
  return ret;
}


/* Sysfs store() method. Calls the store() method
   corresponding to the individual sysfs file */
static ssize_t
l_store(struct kobject *kobj, struct attribute *a,
        const char *buf, size_t count)
{
  int ret;
  struct led_attr *lattr = container_of(a, struct led_attr, attr);

  ret = lattr->store ? lattr->store(buf, count) : -EIO;
  return ret;
}

/* Sysfs operations structure */
static struct sysfs_ops sysfs_ops = {
  .show  = l_show,
  .store = l_store,
};
```

```
/* Attributes of the /sys/class/pardevice/led/control/ kobject.
   Each file in this directory corresponds to one LED. Control
   each LED by writing or reading the associated sysfs file */
static struct attribute *led_attrs[] = {
  &led0.attr,
  &led1.attr,
  &led2.attr,
  &led3.attr,
  &led4.attr,
  &led5.attr,
  &led6.attr,
  &led7.attr,
  NULL
};

/* This describes the kobject. The kobject has 8 files, one
   corresponding to each LED. This representation is called the
   ktype of the kobject */
static struct kobj_type ktype_led = {
  .sysfs_ops = &sysfs_ops,
  .default_attrs = led_attrs,
};

/* Parport methods. We don't have a detach method */
static struct parport_driver led_driver = {
  .name = "led",
  .attach = led_attach,
};

/* Driver Initialization */
int __init
led_init(void)
{
  struct class_device *c_d;

  /* Create the pardevice class - /sys/class/pardevice */
  led_class = class_create(THIS_MODULE, "pardevice");
  if (IS_ERR(led_class)) printk("Bad class create\n");

  /* Create the led class device - /sys/class/pardevice/led/ */
  c_d = class_device_create(led_class, NULL, dev_number,
                                  NULL, DEVICE_NAME);

  /* Register this driver with parport */
  if (parport_register_driver(&led_driver)) {
    printk(KERN_ERR "Bad Parport Register\n");
    return -EIO;
  }

  /* Instantiate a kobject to control each LED
     on the board */

  /* Parent is /sys/class/pardevice/led/ */
  kobj.parent = &c_d->kobj;
  /* The sysfs file corresponding to kobj is
     /sys/class/pardevice/led/control/ */
  strlcpy(kobj.name, "control", KOBJ_NAME_LEN);

  /* Description of the kobject. Specifies the list of attribute
```

```
     files in /sys/class/pardevice/led/control/ */
  kobj.ktype = &ktype_led;

  /* Register the kobject */
  kobject_register(&kobj);

  printk("LED Driver Initialized.\n");
  return 0;
}

/* Driver Exit */
void
led_cleanup(void)
{
  /* Unregister kobject corresponding to
     /sys/class/pardevice/led/control */
  kobject_unregister(&kobj);

  /* Destroy class device corresponding to
     /sys/class/pardevice/led/ */
  class_device_destroy(led_class, MKDEV(MAJOR(dev_number), 0));

  /* Destroy /sys/class/pardevice */
  class_destroy(led_class);

  return;
}

module_init(led_init);
module_exit(led_cleanup);

MODULE_LICENSE("GPL");
```

The macro definition of `glow_show_led()` in Listing 5.7 uses a technique popular in kernel source files to compactly define several similar functions. The definition produces `read()` and `write()` methods (called `show()` and `store()` in sysfs terminology) attached to eight */sys* files, one per LED on the board. Thus, `glow_show_led(0)` attaches `glow_led_0()` and `show_led_0()` to the */sys* file corresponding to the first LED. These functions are respectively responsible for glowing/extinguishing the first LED and reading its status. `##` glues a value to a string, so `glow_led_##number` translates to `glow_led_0()` when the compiler processes the statement, `glow_show_led(0)`.

This sysfs-aware version of the driver uses a kobject to represent a "control" abstraction, which emulates a software knob to control the LEDs. Each kobject is represented by a directory name in sysfs, so `kobject_register()` in Listing 5.7 results in the creation of the */sys/class/pardevice/led/control/* directory.

A ktype describes a kobject. The "control" kobject is described via the `ktype_led` structure, which contains a pointer to the attribute array, `led_attrs[]`. This array contains the addresses of the device attributes of each LED. The attributes of each LED are tied together by the statement:

```
static struct led_attr led##number =
__ATTR(led##number, 0644, show_led_##number, glow_led_##number);
```

This results in instantiating the control file for each LED, */sys/class/pardevice/led/control/ledX*, where *X* is the LED number. To change the state of *ledX*, echo a `1` (or a `0`) to the corresponding control file. To glow the first LED on the board, do this:

```
bash> echo 1 > /sys/class/pardevice/led/control/led0
```

During module exit, the driver unregisters the kobjects and classes using `kobject_unregister()`, `class_device_destroy()`, and `class_destroy()`.

Listing 7.2 in Chapter 7, "Input Drivers," uses another route to create files in sysfs.

Writing a char driver is no longer as simple as it used to be in the days of the 2.4 kernel. To develop the simple LED driver above, we used half a dozen abstractions: cdev, sysfs, kobjects, classes, class device, and parport. The abstractions, however, bring several advantages to the table such as bug-free building blocks, code reuse, and elegant design.

RTC Subsystem

RTC support in the kernel is architected into two layers: a hardware-independent top-layer char driver that implements the kernel RTC API, and a hardware-dependent bottom-layer driver that communicates with the underlying bus. The RTC API, specified in *Documentation/rtc.txt*, is a set of standard ioctls that conforming applications such as hwclock leverage by operating on */dev/rtc*. The API also specifies attributes in sysfs (*/sys/class/rtc/*) and procfs (*/proc/driver/rtc*). The RTC API guarantees that user space tools are independent of the underlying platform and the RTC chip. The bottom-layer RTC driver is bus-specific. The embedded device discussed in the section "Device Example: Real Time Clock" in Chapter 8, "The Inter-Integrated Circuit Protocol," has an RTC chip connected to the I^2C bus, which is driven by an I^2C client driver.

The kernel has a dedicated RTC subsystem that provides the top-layer char driver and a core infrastructure that bottom-layer RTC drivers can use to tie in with the top layer. The main components of this infrastructure are the `rtc_class_ops` structure and the registration functions, `rtc_device_[register|unregister]()`. Bottom-layer RTC drivers scattered under different bus-specific directories are being unified with this subsystem under *drivers/rtc/*.

The RTC subsystem allows the possibility that a system can have more than one RTC. It does this by exporting multiple interfaces, */dev/rtcN* and */sys/class/rtc/rtcN*, where *N* is the number of RTCs on your system. Some embedded systems, for example, have two RTCs: one built in to the microcontroller to support sophisticated operations such as periodic interrupt generation, and another no-frills low-power battery-backed external RTC for timekeeping. Because RTC-aware applications operate over */dev/rtc*, set up a symbolic link so that one of the created */dev/rtcX* nodes can be accessed as */dev/rtc*.

To enable the RTC subsystem, turn on `CONFIG_RTC_CLASS` during kernel configuration.

The Legacy PC RTC Driver

On PC systems, you have the option of bypassing the RTC subsystem by using the legacy RTC driver, *drivers/char/rtc.c.* This driver provides top and bottom layers for the RTC on PC-compatible systems and exports */dev/rtc* and */proc/driver/rtc* to user applications. To enable this driver, turn on `CONFIG_RTC` during kernel configuration.

Pseudo Char Drivers

Several commonly used kernel facilities are not connected with any physical hardware, and these are elegantly implemented as char devices. The *null* sink, the perpetual *zero* source, and the kernel random number generator are treated as virtual devices and are accessed using *pseudo* char device drivers.

The */dev/null* char device sinks data that you don't want to display on your screen. So if you need to check out source files from a *Concurrent Versioning System* (CVS) repository without spewing filenames all over the screen, do this:

```
bash> cvs co kernel > /dev/null
```

This redirects command output to the write entry point belonging to the */dev/null* driver. The driver's `read()` and `write()` methods simply return success ignoring the contents of the input and output buffers, respectively.

If you want to fill an image file with zeros, call upon */dev/zero* to come to your service:

```
bash> dd if=/dev/zero of=file.img bs=1024 count=1024
```

This sources a stream of zeros from the `read()` method belonging to the */dev/zero* driver. The driver has no `write()` method.

The kernel has a built-in random number generator. For the benefit of kernel users who desire to use random sequences, the random number generator exports APIs such as `get_random_bytes()`. For user mode programs, it exports two char interfaces: */dev/random* and */dev/urandom*. The quality of randomness is higher for reads from */dev/random* compared to that from */dev/urandom*. When a user program reads from */dev/random*, it gets strong (or true) random numbers, but reads from */dev/urandom* yield pseudo random numbers. The */dev/random* driver does not use formulae to generate strong random numbers. Instead, it gathers "environmental noise" (interval between interrupts, key clicks, and so on) for maintaining a reservoir of disorder (called an *entropy pool*) that seeds the random stream. To see the kernel's *input* subsystem (discussed in Chapter 7) contributing to the entropy pool when it detects a keyboard press or mouse movement, look at `input_event()` defined in *drivers/input/input.c:*

```
void
input_event(struct input_dev *dev, unsigned int type,
            unsigned int code, int value)
{
  /* ... */
  add_input_randomness(type, code, value); /* Contribute to entropy
                                              pool */
  /* ... */
}
```

To see how the core interrupt handling layer contributes inter-interrupt periods to the entropy pool, look at `handle_IRQ_event()` defined in *kernel/irq/handle.c:*

```
irqreturn_t handle_IRQ_event(unsigned int irq,
                             struct irqaction *action)
{
  /* ... */
  if (status & IRQF_SAMPLE_RANDOM)
```

```
    add_interrupt_randomness(irq); /* Contribute to entropy pool */
  /* ... */
}
```

The generation of strongly random numbers depends on the size of the entropy pool:

```
bash> od -x /dev/random
0000000 7331 9028 7c89 4791 7f64 3deb 86b3 7564
0000020 ebb9 e806 221a b8f9 af12 cb30 9a0e cc28
0000040 68d8 0bbf 68a4 0898 528e 1557 d8b3 57ec
0000060 b01d 8714 b1e1 19b9 0a86 9f60 646c c269
```

The output stops after a few lines, signaling that the entropy pool is exhausted. To replenish the entropy pool and restart the random stream, jab the keyboard several times after switching to an unused terminal or push the mouse around the screen.

A dump of */dev/ urandom*, however, produces a continuous pseudo random stream that never stops.

/dev/mem and */dev/ kmem* are classic pseudo char devices that are tools that let you peek inside system memory. These char nodes export raw interfaces connected to physical memory and kernel virtual memory, respectively. To manipulate system memory, you may `mmap()` these nodes and operate on the returned regions. As an exercise, change the hostname of your system by accessing */dev/mem*.

All the char devices discussed in this section (null, zero, random, urandom, mem, and kmem) have different minor numbers but the same statically assigned major number, 1. Look at *drivers/char/mem.c* and *drivers/char/random.c* for their implementation. Two other pseudo drivers belong to the same major number family: */dev/full*, which emulates an always full device; and */dev/port*, which peeks at system I/O ports. We use the latter in Chapter 19.

Misc Drivers

Misc (or miscellaneous) drivers are simple char drivers that share certain common characteristics. The kernel abstracts these commonalities into an API (implemented in *drivers/char/misc.c*), and this simplifies the way these drivers are initialized. All misc devices are assigned a major number of 10, but each can choose a single minor number. So, if a char driver needs to drive multiple devices as in the CMOS example discussed earlier, it's probably not a candidate for being a misc driver.

Consider the sequence of initialization steps that a char driver performs:

- Allocates major/minor numbers via `alloc_chrdev_region()` and friends
- Creates */dev* and */sys* nodes using `class_device_create()`
- Registers itself as a char driver using `cdev_init()` and `cdev_add()`

A misc driver accomplishes all this with a single call to `misc_register()`:

```
static struct miscdevice mydrv_dev = {
  MYDRV_MINOR,
  "mydrv",
  &mydrv_fops
};

misc_register(&mydrv_dev);
```

In the preceding example, `MYDRV_MINOR` is the minor number that you want to statically assign to your misc driver. You may also request a minor number to be dynamically assigned by specifying `MISC_DYNAMIC_MINOR` rather than `MYDRV_MINOR` in the `mydrv_dev` structure.

Each misc driver automatically appears under */sys/class/misc/* without explicit effort from the driver writer. Because misc drivers are char drivers, the earlier discussion on char driver entry points hold for misc drivers, too. Let's now look at an example misc driver.

Device Example: Watchdog Timer

A watchdog's function is to return an unresponsive system to operational state. It does this by periodically checking the system's pulse and issuing a reset[4] if it can't detect any. Application software is responsible for registering this pulse (or "heartbeat") by periodically strobing (or "petting") the watchdog using the services of a watchdog device driver. Most embedded controllers support internal watchdog modules. External watchdog chips are also available. An example is the Netwinder W83977AF chip.

[4] A watchdog may issue audible beeps rather than a system reset. An example scenario is when a timeout occurs due to a power supply problem, assuming that the watchdog circuit is backed up using a battery or a super capacitor.

Linux watchdog drivers are implemented as misc drivers and live inside *drivers/char/watchdog/*. Watchdog drivers, like RTC drivers, export a standard device interface to user land, so conforming applications are rendered independent of the internals of watchdog hardware. This API is specified in *Documentation/watchdog/watchdog-api.txt* in the kernel source tree. Programs that desire the services of a

watchdog operate on */dev/watchdog*, a device node having a misc minor number of 130.

Listing 5.9 implements a device driver for a fictitious watchdog module built in to an embedded controller. The example watchdog contains two main registers as shown in Table 5.2: a service register (`WD_SERVICE_REGISTER`) and a control register (`WD_CONTROL_REGISTER`). To pet the watchdog, the driver writes a specific sequence (`0xABCD` in this case) to the service register. To program watchdog timeout, the driver writes to specified bit positions in the control register.

Table 5.2. Register Layout on the Watchdog Module

Register Name	Description
`WD_SERVICE_REGISTER`	Write a specific sequence to this register to pet the watchdog.
`WD_CONTROL_REGISTER`	Write the watchdog timeout to this register.

Strobing the watchdog is usually done from user space because the goal of having a watchdog is to detect and respond to both application and kernel hangs. A critical application[5] such as the graphics engine in Listing 5.10 opens the watchdog driver in Listing 5.9 and periodically writes to it. If no write occurs within the watchdog timeout due to an application hang or a kernel crash, the watchdog triggers a system reset. In the case of Listing 5.10, the watchdog will reboot the system if

[5] If you need to monitor the health of several applications, you may implement a multiplexer in the watchdog device driver. If any one of the processes that open the driver becomes unresponsive, the watchdog attempts to self-correct the system.

- The application hangs inside `process_graphics()`
- The kernel, and consequently the application, dies

The watchdog starts ticking when an application opens */dev/watchdog.* Closing this device node stops the watchdog unless you set `CONFIG_WATCHDOG_NOWAYOUT` during kernel configuration. Setting this option helps you tide over the possibility that the watchdog monitoring process (such as Listing 5.10) gets killed by a signal while the system continues running.

Listing 5.9. An Example Watchdog Driver

```
Code View:
#include <linux/miscdevice.h>
#include <linux/watchdog.h>

#define DEFAULT_WATCHDOG_TIMEOUT 10  /* 10-second timeout */
#define TIMEOUT_SHIFT             5  /* To get to the timeout field
                                        in WD_CONTROL_REGISTER */
#define WENABLE_SHIFT             3  /* To get to the
                                        watchdog-enable field in
                                        WD_CONTROL_REGISTER */

/* Misc structure */
static struct miscdevice my_wdt_dev = {
 .minor = WATCHDOG_MINOR, /* defined as 130 in
                             include/linux/miscdevice.h*/
```

```
 .name = "watchdog",      /* /dev/watchdog */
 .fops = &my_wdt_dog  /* Watchdog driver entry points */
};

/* Driver methods */
struct file_operations my_wdt_dog = {
.owner = THIS_MODULE,
.open = my_wdt_open,
.release = my_wdt_close,
.write = my_wdt_write,
.ioctl = my_wdt_ioctl
}

/* Module Initialization */
static int __init
my_wdt_init(void)
{
  /* ... */
  misc_register(&my_wdt_dev);
  /* ... */
}
/* Open watchdog */
static void
my_wdt_open(struct inode *inode, struct file *file)
{
  /* Set the timeout and enable the watchdog */
  WD_CONTROL_REGISTER |= DEFAULT_WATCHDOG_TIMEOUT << TIMEOUT_SHIFT;
  WD_CONTROL_REGISTER |= 1 << WENABLE_SHIFT;
}

/* Close watchdog */
static int
my_wdt_close(struct inode *inode, struct file *file)
{
  /* If CONFIG_WATCHDOG_NOWAYOUT is chosen during kernel
     configuration, do not disable the watchdog even if the
     application desires to close it */
#ifndef CONFIG_WATCHDOG_NOWAYOUT
  /* Disable watchdog */
  WD_CONTROL_REGISTER &= ~(1 << WENABLE_SHIFT);
#endif
  return 0;
}

/* Pet the dog */
static ssize_t
my_wdt_write(struct file *file, const char *data,
             size_t len, loff_t *ppose)
{
  /* Pet the dog by writing a specified sequence of bytes to the
     watchdog service register */
  WD_SERVICE_REGISTER = 0xABCD;
}

/* Ioctl method. Look at Documentation/watchdog/watchdog-api.txt
   for the full list of ioctl commands. This is standard across
   watchdog drivers, so conforming applications are rendered
   hardware-independent */
static int
```

```
my_wdt_ioctl(struct inode *inode, struct file *file,
             unsigned int cmd, unsigned long arg)
{
   /* ... */
   switch (cmd) {
     case WDIOC_KEEPALIVE:
       /* Write to the watchdog. Applications can invoke
          this ioctl instead of writing to the device */
       WD_SERVICE_REGISTER = 0xABCD;
       break;
     case WDIOC_SETTIMEOUT:
        copy_from_user(&timeout, (int *)arg, sizeof(int));

       /* Set the timeout that defines unresponsiveness by
          writing to the watchdog control register */
        WD_CONTROL_REGISTER = timeout << TIMEOUT_BITS;
       break;
     case WDIOC_GETTIMEOUT:
       /* Get the currently set timeout from the watchdog */
       /* ... */
       break;
     default:
       return –ENOTTY;
   }
}

/* Module Exit */
static void __exit
my_wdt_exit(void)
{
  /* ... */
  misc_deregister(&my_wdt_dev);
  /* ... */
}

module_init(my_wdt_init);
module_exit(my_wdt_exit);
```

Listing 5.10. A Watchdog User

```
#include <fcntl.h>
#include <asm/types.h>
#include <linux/watchdog.h>

int
main()
{
  int new_timeout;

  int wfd = open("/dev/watchdog", O_WRONLY);

  /* Set the watchdog timeout to 20 seconds */
  new_timeout = 20;
  ioctl(fd, WDIOC_SETTIMEOUT, &new_timeout);

  while (1) {
    /* Graphics processing */
    process_graphics();
    /* Pet the watchdog */
    ioctl(fd, WDIOC_KEEPALIVE, 0);
    /* Or instead do: write(wfd, "\0", 1); */
    fsync(wfd);
  }
}
```

External Watchdogs

To ensure that the system attempts to recover even in the face of processor failures, some regulatory bodies stipulate the use of an external watchdog chip, even if the main processor has a sophisticated built-in watchdog module such as the one in our example. Because of this requirement, embedded devices sometimes use an inexpensive no-frill watchdog chip (such as MAX6730 from Maxim) that is based on simple hard-wired logic rather than a register interface. The watchdog asserts a *reset* pin if no voltage pulse is detected on an *input* pin within a fixed reset timeout. The reset pin is connected to the reset logic of the processor, and the input pin is wired to a processor GPIO port. All that software has to do to prevent reset is to periodically pulse the watchdog's input pin within the chip's reset timeout. If you are writing a driver for such a device, the `ioctl()` method is not relevant. The driver's `write()` method pulses the watchdog's input pin whenever application software writes to the associated device node. To aid manufacturing and field diagnostics, the watchdog is wired such that it can be disabled by wiggling a processor GPIO pin.

Such chips usually allow a large initial timeout to account for boot time, followed by shorter reset timeouts.

For platforms that do not support a hardware watchdog module, the kernel implements a software watchdog, also called a *softdog.* The softdog driver, *drivers/char/watchdog/softdog.c*, is a pseudo misc driver because it does not operate on real hardware. The softdog driver has to perform two tasks that a watchdog driver doesn't have to do, which the latter accomplishes in hardware:

- Implement a timeout mechanism

- Initiate a soft reboot if the system isn't healthy

This is done by delaying the execution of a timer handler whenever an application writes to the softdog. If no write occurs to the softdog within a timeout, the timer handler fires and reboots the system.

A related support in 2.6 kernels is the sensing of *soft lockups*, which are instances when scheduling does not occur for 10 or more seconds. A kernel thread *watchdog/N*, where *N* is the CPU number, touches a per-CPU timestamp every second. If the thread doesn't touch the timestamp for more than 10 seconds, the system is deemed to have locked up. Soft lockup detection (implemented in *kernel/softlockup.c*) will aid us while debugging a kernel crash in the section "Kdump" in Chapter 21, "Debugging Device Drivers."

There are several more misc drivers in the kernel. The Qtronix infrared keyboard driver, *drivers/char/qtronix.c*, is another example of a char driver that has a misc form factor. Do a grep on `misc_register()` in the *drivers/char/* directory to find other misc device drivers present in the kernel.

Character Caveats

Driver methods, and, hence, the associated system calls issued by user applications, may fail or partially succeed. Your application has to factor this in to avoid unpleasant surprises. Let's look at some common pitfalls:

- An `open()` call may fail for several reasons. Some char drivers support only a single user at a time, so they fail with `-EBUSY` if an application attempts to open a device that is already in use. If a printer is out of paper, the driver fails with `-ENOSPC` if you issue a device `open()`.

- A successful `read()` or `write()` can return anything between 1 byte and the number of bytes requested, so your application needs sufficient logic to handle this.

- A `select()` call returns success even if a single byte of data is ready to be read or written.

- Some char devices such as mice and touch screens are input-only, so their drivers will not support the write method family (`write()/aio_write()/fsync()`). Other devices such as printers are output-only, and their drivers will not support the read method family (`read()/aio_read()`). Also, many char driver methods are optional, so all methods will not be present in all drivers. When a method is absent, the corresponding system call fails.

Looking at the Sources

Char drivers do not exclusively live in the *drivers/char/* directory. Here are some examples of "super" char drivers that merit special treatment and directories:

- Serial drivers are char drivers that manage your computer's serial port. However, they are much more than simple char drivers and reside separately in the *drivers/serial/* directory. The next chapter discusses serial drivers.

- Input drivers are responsible for devices such as keyboards, mice, and joysticks. They live in a separate source directory, *drivers/input/* and, hence, get a distinct chapter, Chapter 7.

- Frame buffers (*/dev/fb/**) offer access to video memory, the way */dev/mem* exports access to system memory. Chapter 12, "Video Drivers," looks at frame buffer drivers.

- Some device classes support a minority of hardware possessing a char interface. For example, SCSI devices are generally block devices, but a SCSI tape is a char device.

- Some subsystems export additional char interfaces that present a raw device model to user space. The MTD subsystem is generally used for emulating a disk on top of diverse types of flash memory, but some applications might be better served if they are provided with a raw view of the underlying flash memory. This is done by the MTD char driver, *drivers/mtd/mtdchar.c*, which is discussed in Chapter 17, "Memory Technology Devices."

- Certain kernel layers provide hooks for implementing user-space device drivers by exporting suitable char interfaces. Applications can directly access the innards of the device via these interfaces. One example is the generic SCSI driver *drivers/scsi/sg.c* used to implement user space device drivers for SCSI scanners and CD drives. Another example is the I^2C device interface, *i2c-dev.* Such char interfaces are explained in Chapter 19.

Meanwhile, run a `grep -r` on `register_chrdev` in the *drivers/* directory to get an idea of the popularity of char drivers in the kernel.

Table 5.3 contains a summary of the main data structures used in this chapter and the location of their definitions in the source tree. Table 5.4 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 5.3. Summary of Data Structures

Data Structure	Location	Description
`cdev`	*include/linux/cdev.h*	Kernel abstraction of a char device
`file_operations`	*include/linux/fs.h*	Char driver methods
`dev_t`	*include/linux/types.h*	Device major/minor numbers

Data Structure	Location	Description
`poll_table`	*include/linux/poll.h*	A table of wait queues owned by drivers that are being polled for data
`pardevice`	*include/linux/parport.h*	Kernel abstraction of a parallel port device
`rtc_class_ops`	*include/linux/rtc.h*	Communication interface between top layer and bottom layer RTC drivers
`miscdevice`	*include/linux/miscdevice.h*	Representation of a misc device

Table 5.4. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`alloc_chrdev_region()`	*fs/char_dev.c*	Requests dynamic allocation of a device major number
`unregister_chrdev_region()`	*fs/char_dev.c*	Reverse of `alloc_chrdev_region()`
`cdev_init()`	*fs/char_dev.c*	Connects char driver methods with the associated `cdev`
`cdev_add()`	*fs/char_dev.c*	Associates a device number with a `cdev`
`cdev_del()`	*fs/char_dev.c*	Removes a `cdev`
`container_of()`	*include/linux/kernel.h*	From a structure member, gets the address of its containing structure
`copy_from_user()`	*arch/x86/lib/usercopy_32.c (For i386)*	Copies data from user space to kernel space
`copy_to_user()`	*arch/x86/lib/usercopy_32.c (For i386)*	Copies data from kernel space to user space
`likely()` `unlikely()`	*include/linux/compiler.h*	Informs GCC about the possibility of success of the associated conditional evaluation
`request_region()`	*include/linux/ioport.h kernel/resource.c*	Stakes claim to an I/O region
`release_region()`	*include/linux/ioport.h kernel/resource.c*	Relinquishes claim to an I/O region
`in[b|w|l|sn|sl]()` `out[b|w|l|sn|sl]()`	*include/asm-your-arch/io.h*	Family of functions to exchange data with I/O regions
`poll_wait()`	*include/linux/poll.h*	Adds a wait queue to the kernel `poll_table`
`fasync_helper()`	*fs/fcntl.c*	Ensures that if a driver issues a `kill_fasync()`, a `SIGIO` is dispatched to the owning application

Kernel Interface	Location	Description
`kill_fasync()`	*fs/fcntl.c*	Dispatches a `SIGIO` to the owning application
`parport_register_device()`	*drivers/parport/share.c*	Registers a parallel port device with parport
`parport_unregister_device()`	*drivers/parport/share.c*	Unregisters a parallel port device
`parport_register_driver()`	*drivers/parport/share.c*	Registers a parallel port driver with parport
`parport_unregister_driver()`	*drivers/parport/share.c*	Unregisters a parallel port driver
`parport_claim_or_block()`	*drivers/parport/share.c*	Claims a parallel port
`parport_write_data()`	*include/linux/parport.h*	Writes data to a parallel port
`parport_read_data()`	*include/linux/parport.h*	Reads data from a parallel port
`parport_release()`	*drivers/parport/share.c*	Releases a parallel port
`kobject_register()`	*lib/kobject.c*	Registers a kobject and creates associated files in sysfs
`kobject_unregister()`	*lib/kobject.c*	Reverse of `kobject_register()`
`rtc_device_register()`/ `rtc_device_unregister()`	*drivers/rtc/class.c*	Registers/unregisters a bottom-layer driver with the RTC subsystem
`misc_register()`	*drivers/char/misc.c*	Registers a misc driver
`misc_deregister()`	*drivers/char/misc.c*	Unregisters a misc driver

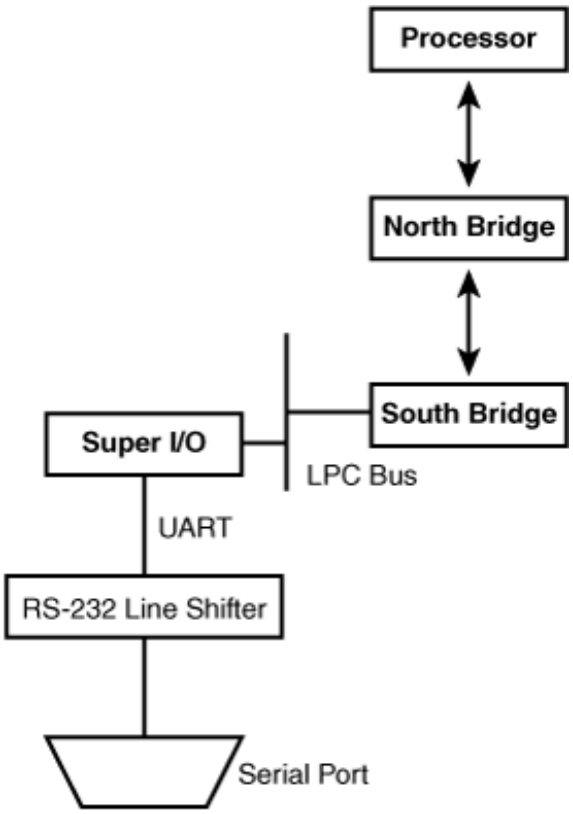
Chapter 6. Serial Drivers

In This Chapter

- Layered Architecture 173
- UART Drivers 176
- TTY Drivers 192
- Line Disciplines 194
- Looking at the Sources 205

The serial port is a basic communication channel used by a slew of technologies and applications. A chip known as the *Universal Asynchronous Receiver Transmitter* (UART) is commonly used to implement serial communication. On PC-compatible hardware, the UART is part of the Super I/O chipset, as shown in Figure 6.1.

Figure 6.1. Connection diagram of the PC serial port.



Though RS-232 communication channels are the common type of serial hardware, the kernel's serial subsystem is architected in a generic manner to serve diverse users. You will touch the serial subsystem if you

- Run a terminal session over an RS-232 serial link
- Connect to the Internet via a dialup, cellular, or software modem
- Interface with devices such as touch controllers, smart cards, Bluetooth chips, or Infrared dongles, which use a serial transport
- Emulate a serial port using a USB-to-serial converter
- Communicate over an RS-485 link, which is a multidrop variant of RS-232 that has larger range and better noise immunity

In this chapter, let's find out how the kernel structures the serial subsystem. We will use the example of a Linux cell phone to learn about low-level UART drivers and the example of a serial touch controller to discover the implementation details of higher-level line disciplines.

The UART usually found on PCs is National Semiconductor's 16550, or compatible chips from other vendors, so you will find references to "16550-type UART" in code and documentation. The 8250 chip is the predecessor of the 16550, so the Linux driver for PC UARTs is named *8250.c*.

Layered Architecture

As you just saw, the users of the serial subsystem are many and different. This has motivated kernel developers to structure a layered serial architecture using the following building blocks:

1. Low-level drivers that worry about the internals of the UART or other underlying serial hardware.

2. A tty driver layer that interfaces with the low-level driver. A tty driver insulates higher layers from the intricacies of the hardware.

3. Line disciplines that "cook" data exchanged with the tty driver. Line disciplines shape the behavior of the serial layer and help reuse lower layers to support different technologies.

To help custom driver implementations, the serial subsystem also provides core APIs that factor commonalities out of these layers.

Figure 6.2 shows the connection between the layers. `N_TTY`, `N_IRDA`, and `N_PPP` are drop-in line disciplines that mold the serial subsystem to respectively support terminals, Infrared, and dialup networking. Figure 6.3 maps the serial subsystem to kernel source files.

Figure 6.2. Connection between the layers in the serial subsystem.

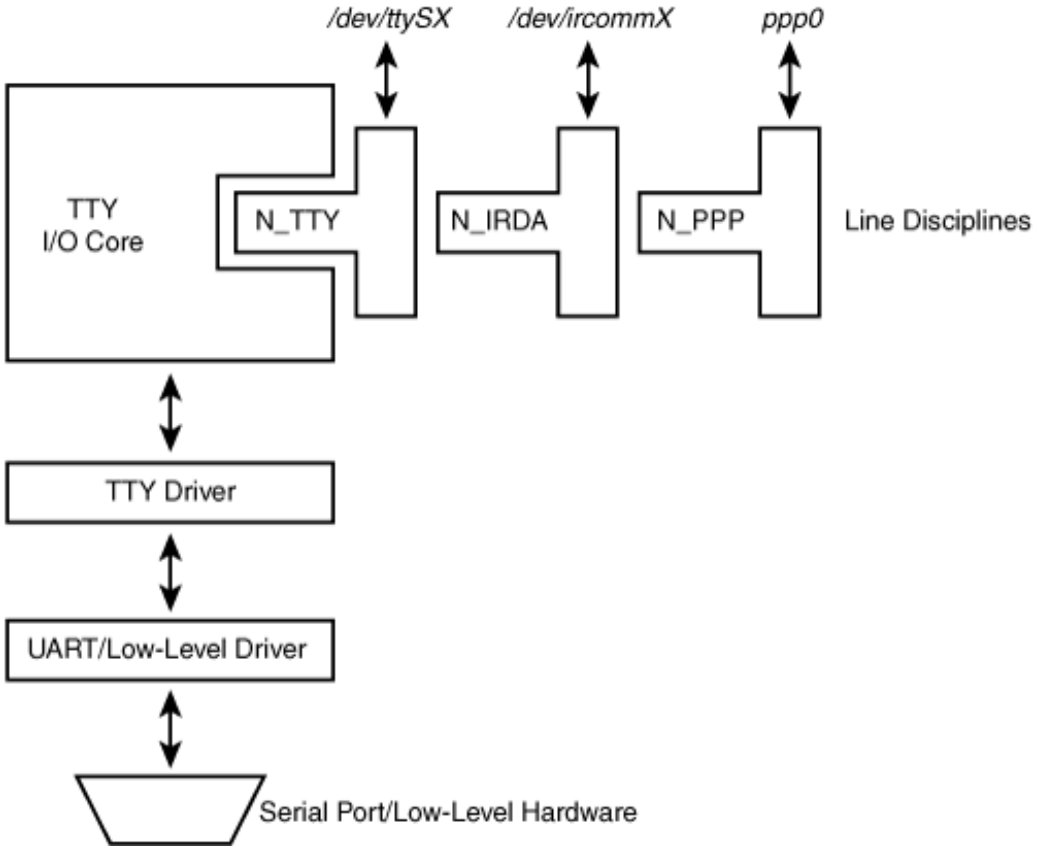
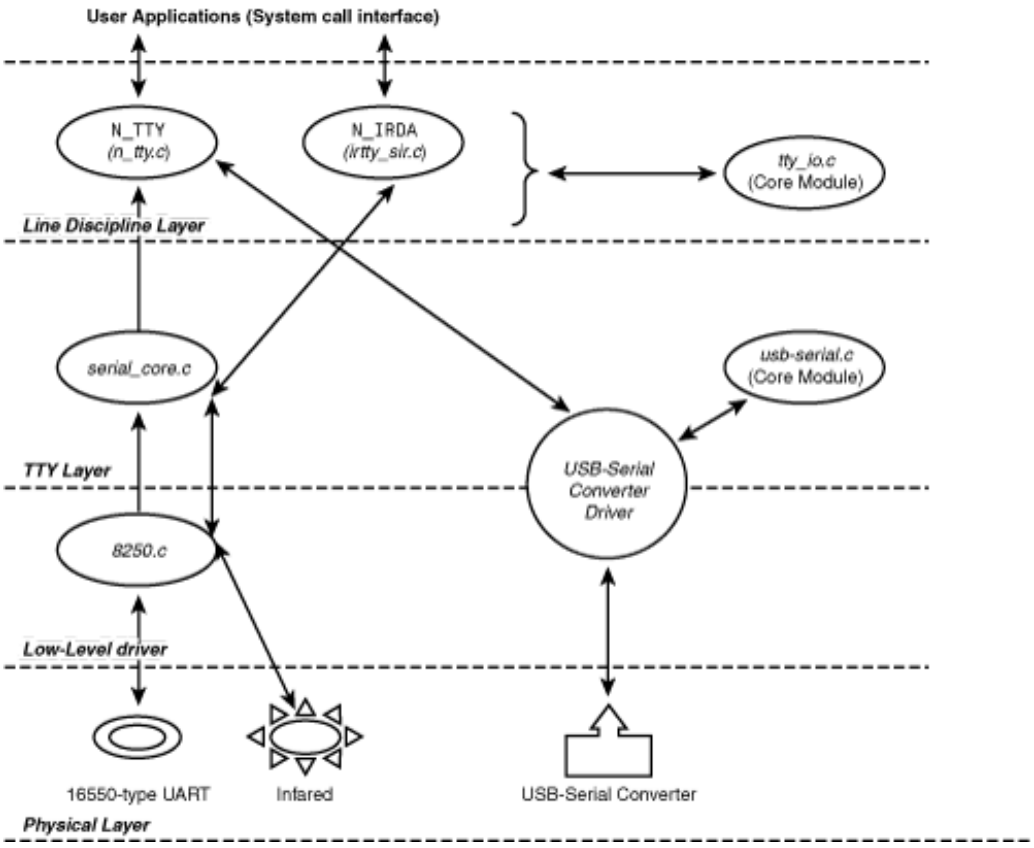


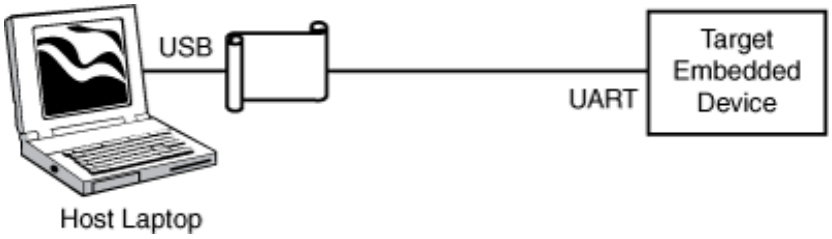
Figure 6.3. Serial layers mapped to kernel sources.

[View full size image]



To illustrate the advantages of a layered serial architecture, let's use an example. Assume that you are using a USB-to-serial adapter to obtain serial capabilities on a laptop that does not have a serial port. One possible scenario is when you are debugging the kernel on a target embedded device from a host laptop using kgdb (kgdb is discussed in Chapter 21, "Debugging Device Drivers"), as shown in Figure 6.4.

Figure 6.4. Using a USB-to-serial converter.



As shown in Figure 6.3, you first need a suitable USB physical layer driver (the USB counterpart of the UART driver) on your host laptop. This is present in the kernel USB subsystem, *drivers/usb/*. Next, you need a tty driver to sit on top of the USB physical layer. The *usbserial* driver (*drivers/usb/serial/usb-serial.c*) is the core layer that implements a generic tty over USB-serial converters. This driver, in tandem with device-specific tty methods registered by the converter driver (*drivers/usb/serial/keyspan.c* if you are using a Keyspan adapter, for example), constitutes the tty layer. Last, but not the least, you need the `N_TTY` line discipline for terminal I/O processing.

The tty driver insulates the line discipline and higher layers from the internals of USB. In fact, the line discipline still thinks it's running over a conventional UART. This is so because the tty driver pulls data from *USB Request Blocks* or URBs (discussed in Chapter 11, "Universal Serial Bus") and encapsulates it in the format expected by the `N_TTY` line discipline. The layered architecture thus renders the implementation simpler—all blocks from the line discipline upward can be reused unchanged.

The preceding example uses a technology-specific tty driver and a generic line discipline. The reverse usage is also common. The Infrared stack, discussed in Chapter 16, "Linux Without Wires," uses a generic tty driver and a technology-specific line discipline called `N_IRDA`.

As you might have noticed in Figure 6.2 and Figure 6.3, although UART drivers are char drivers, they do not directly expose interfaces to kernel system calls like regular char drivers that we saw in the preceding chapter. Rather, UART drivers (like keyboard drivers discussed in the next chapter) service another kernel layer, the tty layer. I/O system calls start their journey above top-level line disciplines and finally ripple down to UART drivers through the tty layer.

In the rest of this chapter, let's take a closer look at the different driver components of the serial layer. We start at the bottom of the serial stack with low-level UART drivers, move on to middle-level tty drivers, and then proceed to top-level line discipline drivers.

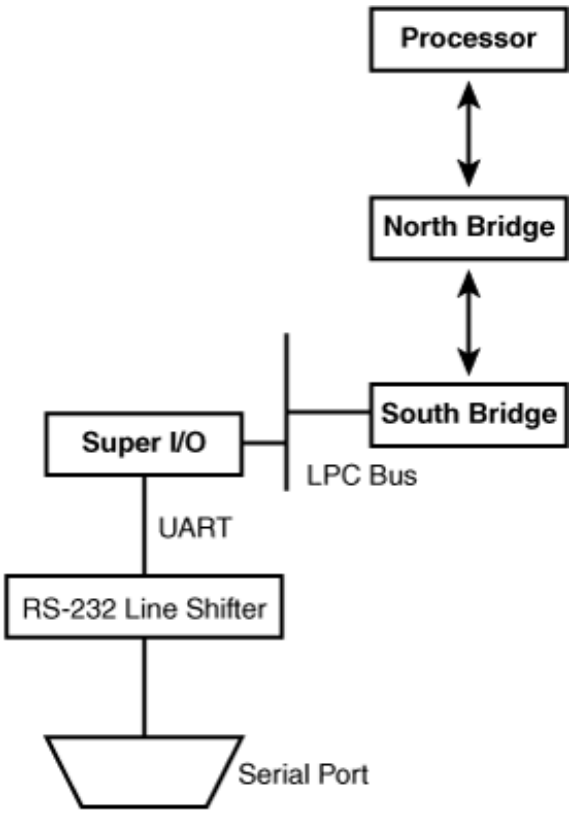
Chapter 6. Serial Drivers

In This Chapter

- Layered Architecture 173
- UART Drivers 176
- TTY Drivers 192
- Line Disciplines 194
- Looking at the Sources 205

The serial port is a basic communication channel used by a slew of technologies and applications. A chip known as the *Universal Asynchronous Receiver Transmitter* (UART) is commonly used to implement serial communication. On PC-compatible hardware, the UART is part of the Super I/O chipset, as shown in Figure 6.1.

Figure 6.1. Connection diagram of the PC serial port.



Though RS-232 communication channels are the common type of serial hardware, the kernel's serial subsystem is architected in a generic manner to serve diverse users. You will touch the serial subsystem if you

- Run a terminal session over an RS-232 serial link
- Connect to the Internet via a dialup, cellular, or software modem
- Interface with devices such as touch controllers, smart cards, Bluetooth chips, or Infrared dongles, which use a serial transport
- Emulate a serial port using a USB-to-serial converter
- Communicate over an RS-485 link, which is a multidrop variant of RS-232 that has larger range and better noise immunity

In this chapter, let's find out how the kernel structures the serial subsystem. We will use the example of a Linux cell phone to learn about low-level UART drivers and the example of a serial touch controller to discover the implementation details of higher-level line disciplines.

The UART usually found on PCs is National Semiconductor's 16550, or compatible chips from other vendors, so you will find references to "16550-type UART" in code and documentation. The 8250 chip is the predecessor of the 16550, so the Linux driver for PC UARTs is named *8250.c*.

Layered Architecture

As you just saw, the users of the serial subsystem are many and different. This has motivated kernel developers to structure a layered serial architecture using the following building blocks:

1. Low-level drivers that worry about the internals of the UART or other underlying serial hardware.

2. A tty driver layer that interfaces with the low-level driver. A tty driver insulates higher layers from the intricacies of the hardware.

3. Line disciplines that "cook" data exchanged with the tty driver. Line disciplines shape the behavior of the serial layer and help reuse lower layers to support different technologies.

To help custom driver implementations, the serial subsystem also provides core APIs that factor commonalities out of these layers.

Figure 6.2 shows the connection between the layers. `N_TTY`, `N_IRDA`, and `N_PPP` are drop-in line disciplines that mold the serial subsystem to respectively support terminals, Infrared, and dialup networking. Figure 6.3 maps the serial subsystem to kernel source files.

Figure 6.2. Connection between the layers in the serial subsystem.

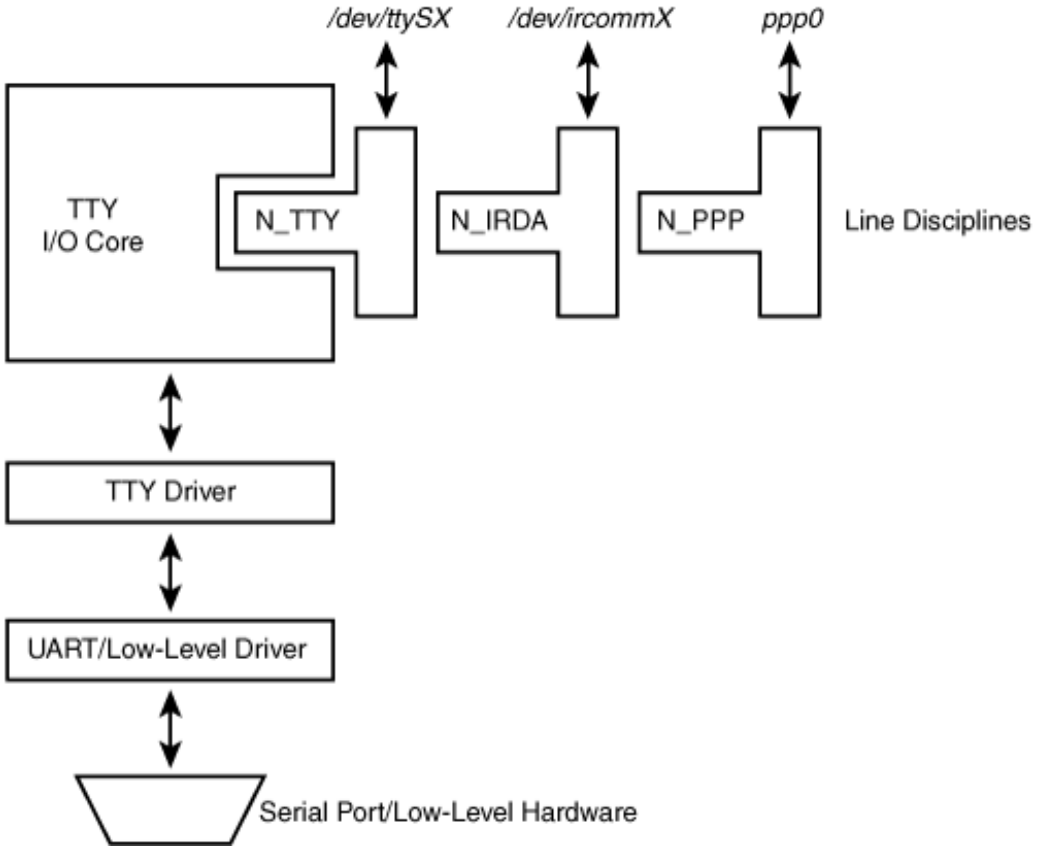
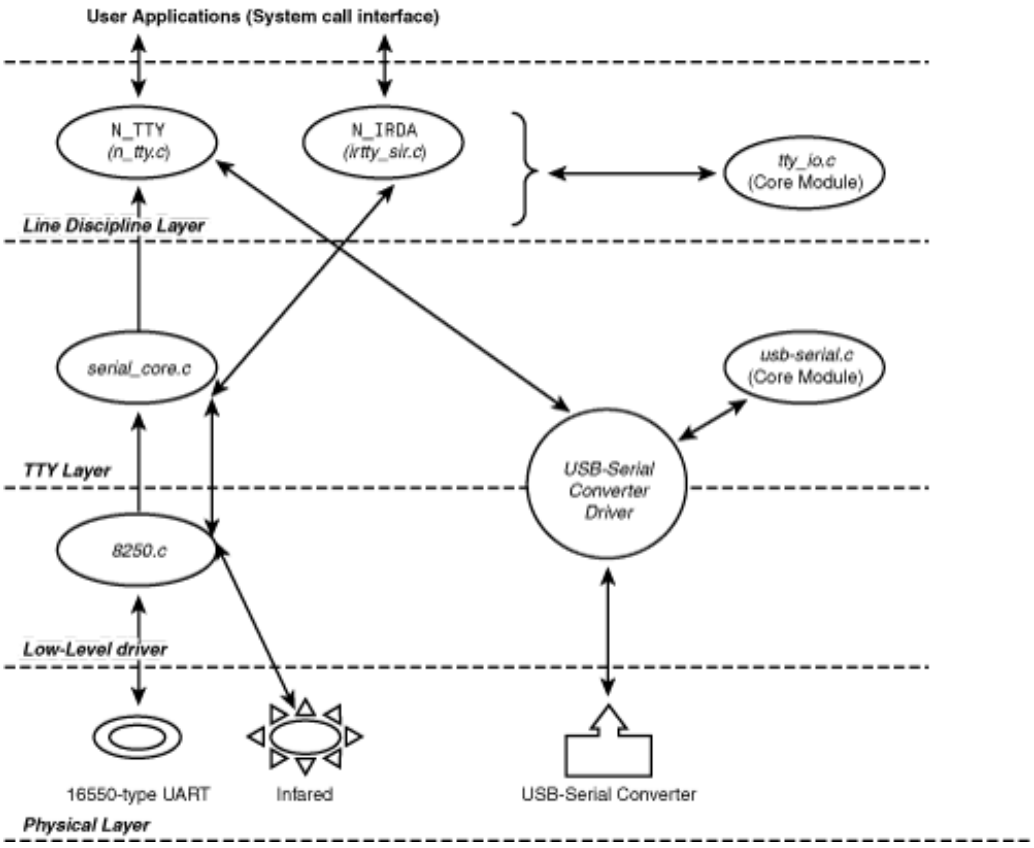


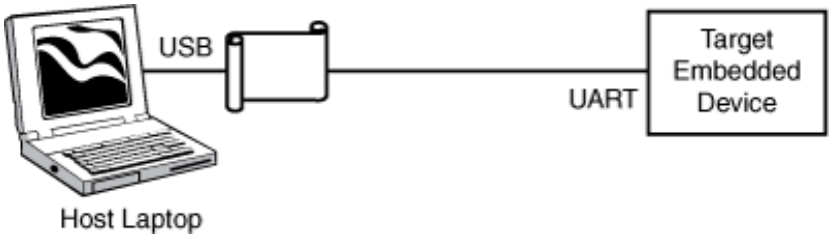
Figure 6.3. Serial layers mapped to kernel sources.

[View full size image]



To illustrate the advantages of a layered serial architecture, let's use an example. Assume that you are using a USB-to-serial adapter to obtain serial capabilities on a laptop that does not have a serial port. One possible scenario is when you are debugging the kernel on a target embedded device from a host laptop using kgdb (kgdb is discussed in Chapter 21, "Debugging Device Drivers"), as shown in Figure 6.4.

Figure 6.4. Using a USB-to-serial converter.



As shown in Figure 6.3, you first need a suitable USB physical layer driver (the USB counterpart of the UART driver) on your host laptop. This is present in the kernel USB subsystem, *drivers/usb/*. Next, you need a tty driver to sit on top of the USB physical layer. The *usbserial* driver (*drivers/usb/serial/usb-serial.c*) is the core layer that implements a generic tty over USB-serial converters. This driver, in tandem with device-specific tty methods registered by the converter driver (*drivers/usb/serial/keyspan.c* if you are using a Keyspan adapter, for example), constitutes the tty layer. Last, but not the least, you need the `N_TTY` line discipline for terminal I/O processing.

The tty driver insulates the line discipline and higher layers from the internals of USB. In fact, the line discipline still thinks it's running over a conventional UART. This is so because the tty driver pulls data from *USB Request Blocks* or URBs (discussed in Chapter 11, "Universal Serial Bus") and encapsulates it in the format expected by the `N_TTY` line discipline. The layered architecture thus renders the implementation simpler—all blocks from the line discipline upward can be reused unchanged.

The preceding example uses a technology-specific tty driver and a generic line discipline. The reverse usage is also common. The Infrared stack, discussed in Chapter 16, "Linux Without Wires," uses a generic tty driver and a technology-specific line discipline called `N_IRDA`.

As you might have noticed in Figure 6.2 and Figure 6.3, although UART drivers are char drivers, they do not directly expose interfaces to kernel system calls like regular char drivers that we saw in the preceding chapter. Rather, UART drivers (like keyboard drivers discussed in the next chapter) service another kernel layer, the tty layer. I/O system calls start their journey above top-level line disciplines and finally ripple down to UART drivers through the tty layer.

In the rest of this chapter, let's take a closer look at the different driver components of the serial layer. We start at the bottom of the serial stack with low-level UART drivers, move on to middle-level tty drivers, and then proceed to top-level line discipline drivers.

UART Drivers

UART drivers revolve around three key data structures. All three are defined in *include/linux/serial_core.h* :

1. The per-UART driver structure, `struct uart_driver` :

```
struct uart_driver {
  struct module  *owner;          /* Module that owns this
                                     struct */
  const char     *driver_name;    /* Name */
  const char     *dev_name;       /* /dev node name
                                     such as ttyS */
  /* ... */
  int             major;          /* Major number */
  int             minor;          /* Minor number */
  /* ... */
  struct tty_driver *tty_driver;  /* tty driver */
};
```

   The comments against each field explain the associated semantics. The `owner` field allows the same benefits as that discussed in the previous chapter for the `file_operations` structure.

2. `struct uart_port` . One instance of this structure exists for each port owned by the UART driver:

```
struct uart_port {
  spinlock_t     lock;             /* port lock */
  unsigned int   iobase;           /* in/out[bwl]*/
  unsigned char  __iomem *membase; /* read/write[bwl]*/
  unsigned int   irq;              /* irq number */
  unsigned int   uartclk;          /* base uart clock */
  unsigned char  fifosize;         /* tx fifo size */
  unsigned char  x_char;           /* xon/xoff flow
                                      control */
  /* ... */
};
```

3. `struct uart_ops` . This is a superset of entry points that each UART driver has to support and describes the operations that can be done on physical hardware. The methods in this structure are invoked by the tty layer:

```
struct uart_ops {
  uint (*tx_empty)(struct uart_port *);     /* Is TX FIFO empty? */
  void (*set_mctrl)(struct uart_port *,
                    unsigned int mctrl);    /* Set modem control params */
  uint (*get_mctrl)(struct uart_port *);    /* Get modem control params */
  void (*stop_tx)(struct uart_port *);      /* Stop xmission */
  void (*start_tx)(struct uart_port *);     /* Start xmission */

  /* ... */
  void (*shutdown)(struct uart_port *);     /* Disable the port */
```

```
    void (*set_termios)(struct uart_port *,
                        struct termios *new,
                        struct termios *old); /* Set terminal interface
                                                 params */
    /* ... */
    void (*config_port)(struct uart_port *,
                        int);                 /* Configure UART port */
    /* ... */
};
```

There are two important steps that a UART driver has to do to tie itself with the kernel:

1.
Register with the serial core by calling

```
uart_register_driver(struct uart_driver *);
```

2.
Invoke `uart_add_one_port(struct uart_driver *` , `struct uart_port *)` to register each individual port that it supports. If your serial hardware is hotplugged, the ports are registered with the core from the entry point that probes the presence of the device. Look at the CardBus Modem driver in Listing 10.4 in Chapter 10 , "Peripheral Component Interconnect," for an example where the serial device is plugged hot. Note that some drivers use the wrapper registration function `serial8250_register_port(struct uart_port *)` , which internally invokes `uart_add_one_port()` .

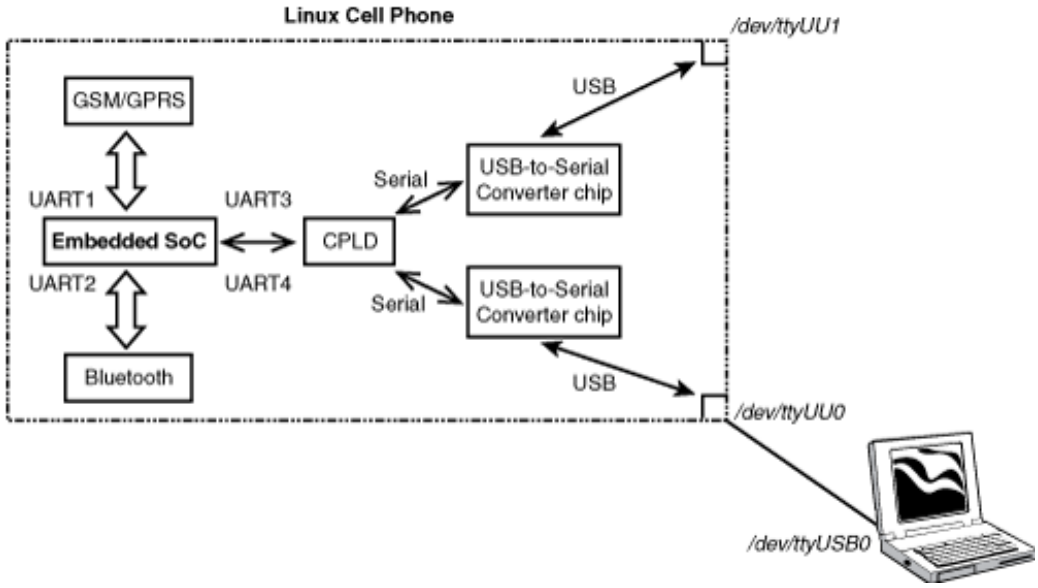
These data structures and registration functions constitute the least common denominator present in all UART drivers. Armed with these structures and routines, let's develop a sample UART driver.

Device Example: Cell Phone

Consider a Linux cell phone built around an embedded *System-on-Chip* (SoC). The SoC has two built-in UARTs, but as shown in Figure 6.5 , both of them are used up, one for communicating with a cellular modem, and the other for interfacing with a Bluetooth chipset. Because there are no free UARTs for debug purposes, the phone uses two USB-to-serial converter chips, one to provide a debug terminal to a PC host, and the other to obtain a spare port. USB-to-serial converters, as you saw earlier in this chapter, let you connect serial devices to your PC via USB. We discuss more on USB-to-serial converters in Chapter 11 .

Figure 6.5. **USB_UART** ports on a Linux cell phone.

[View full size image]



The serial sides of the two USB-to-serial converter chips are connected to the SoC via a *Complex Programmable Logic Device* or CPLD (see the section "CPLD/FPGA " in Chapter 18 , "Embedding Linux"). The CPLD creates two virtual UARTs (or USB_UART s) by providing a three-register interface to access each USB-to-serial converter, as shown in Table 6.1 : a status register, a read-data register, and a write-data register. To write a character to a USB_UART , loop on a bit in the status register that clears when there is space in the chip's internal transmit *first-in first-out* (FIFO) memory and then write the byte to the write-data register. To read a character, wait until a specified bit in the status register shows presence of data in the receive FIFO and then read from the read-data register.

UU_STATUS_REGISTER
Bits to check whether the transmit FIFO is full or whether the receive FIFO is empty
0x0
UU_READ_DATA_REGISTER
Read a character from the USB_UART
0x1
UU_WRITE_DATA_REGISTER
Write a character to the USB_UART
0x2

Table 6.1. Register Layout of the **USB_UART**

Register Name	Description	Offset from **USB_UART** Memory Base

At the PC end, use the appropriate Linux *usbserial* driver (for example, *drivers/usb/serial/ftdi_sio.c* if you are using an FT232AM chip on the cell phone) to create and manage */dev/ttyUSBX* device nodes that correspond to the USB-serial ports. You may run terminal emulators such as *minicom* over one of these device nodes to obtain a console or debug terminal from the cell phone. At the cell phone end, we have to implement a UART driver for the USB_UART s. This driver creates and manages */dev/ttyUUX* nodes that are responsible for communication at the device side of the link.

The cell phone shown in Figure 6.5 can act as an intelligent gateway for Bluetooth devices—to the GSM network and, hence, to the Internet. The phone can, for example, ferry data from your Bluetooth blood pressure monitor to your health-care provider's server on the Internet. Or it can alert a doctor if it detects a problem while communicating with your Bluetooth-enabled heart-rate monitor. The Bluetooth

MP3 player used in Chapter 13 , "Audio Drivers," and the Bluetooth pill dispenser used in Chapter 16 are also examples of devices that can use the Linux cell phone to get Internet-enabled.

Listing 6.1 implements the USB_UART driver. It's implemented as a *platform driver.* A platform is a pseudo bus usually used to tie lightweight devices integrated into SoCs, with the Linux device model. A platform consists of

1. A platform device. The architecture-specific setup code adds the platform device using `platform_device_register()` or its simpler version, `platform_device_register_simple()` . You may also register multiple platform devices at one shot using `platform_add_devices()` . The `platform_device` structure defined in *include/linux/platform_device.h* represents a platform device:

   ```
   struct platform_device {
     const char *name;  /* Device Name */
     u32 id;            /* Use this field to register multiple
                           instances of a platform device. In
                           this example, the two USB_UARTs
                           have different IDs. */
     struct device dev; /* Contains a release() method and
                           platform data */
     /* ... */
   };
   ```

2. A platform driver. The platform driver registers itself into the platform using `platform_driver_register()` . The `platform_driver` structure, also defined in *include/linux/platform_device.h* , represents a platform driver:

   ```
   struct platform_driver {
     int (*probe)(struct platform_device *); /*Probe method*/
     int (*remove)(struct platform_device *);/*Remove method*/
     /* ... */
     /* The name field in the following structure should match
        the name field in the associated platform_device
        structure */
     struct device_driver driver;
   };
   ```

See *Documentation/driver-model/platform.txt* for more on platform devices and drivers. For simplicity, our sample driver registers both the platform device and the platform driver.

During initialization, the USB_UART driver first registers itself with the serial core using `uart_register_driver()` . When this is done, you will find a new line starting with `usb_uart` in */proc/tty/drivers* . Next, the driver registers two platform devices (one per USB_UART) using `platform_device_register_simple()` . As mentioned earlier, platform device registrations are usually done during boot-time board setup. Following this, the driver registers platform driver entry points (`probe()` , `remove()` , `suspend()` , and `resume()`) using `platform_driver_register()` . The USB_UART platform driver ties into both the above platform devices and has a matching name (`usb_uart`). After this step, you will see two new directories appearing in sysfs, each corresponding to a USB_UART port: */sys/devices/platform/usb_uart.0/* and */sys/devices/platform/usb_uart.1/* .

Because the Linux device layer now detects a platform driver matching the name of the registered USB_UART platform devices, it invokes the `probe()` entry point[1] (`usb_uart_probe()`) belonging to the platform driver, once for each USB_UART . The probe entry point adds the associated USB_UART port using `uart_add_one_port()`

. This triggers invocation of the `config_port()` entry point (part of the `uart_ops` structure discussed earlier) that claims and maps the `USB_UART` register space. If both `USB_UART` ports are successfully added, the serial core emits the following kernel messages:

[1] Such platform devices usually cannot be hotplugged. This invocation semantics of the `probe()` method is different from what you will learn in later chapters for hotpluggable devices such as PCMCIA, PCI, and USB, but the similar structure of driver entry points helps the Linux device model to have a uniform and consistent view of all devices.

```
ttyUU0 at MMIO 0xe8000000 (irq = 3) is a USB_UART
ttyUU1 at MMIO 0xe9000000 (irq = 4) is a USB_UART
```

Claiming the IRQ, however, is deferred until an application opens the `USB_UART` port. The IRQ is freed when the application closes the `USB_UART` . Table 6.2 traces the driver's code path for claiming and freeing memory regions and IRQs.

Module Insert
`usb_uart_init()`
`uart_register_driver()`
`usb_uart_probe()`
`uart_add_one_port()`
`usb_uart_config_port()`
`request_mem_region()`
Module Unload
`usb_uart_exit()`
`usb_unregister_driver()`
`usb_uart_remove()`
`uart_remove_one_port()`
`usb_uart_release_port()`
`release_mem_region()`
Open */dev/ttyUUX*
`usb_uart_startup()`
`request_irq()`
Close */dev/ttyUUX*
`usb_uart_shutdown()`
`free_irq()`

Table 6.2. Claiming and Freeing Memory and IRQ Resources

In the transmit path, the driver collects egress data from the circular buffer associated with the UART port. Data is present in `port->info->xmit.buf[port->info->xmit.tail]` as is evident from the UART driver's `start_tx()` entry point, `usb_uart_start_tx()` .

In the receive path, the driver pushes data collected from the `USB_UART` to the associated tty driver using `tty_insert_flip_char()` and `tty_flip_buffer_push()` . This is done in the receive interrupt handler, `usb_uart_rxint()` . Revisit this routine after reading the next section, "TTY Drivers ."

Listing 6.1 uses comments to explain the purpose of driver entry points and their operation. It leaves some of the entry points in the `uart_ops` structure unimplemented to cut out extra detail.

Listing 6.1. **USB_UART** Driver for the Linux Cell Phone

Code View:

```
#include <linux/console.h>
#include <linux/platform_device.h>
#include <linux/tty.h>
#include <linux/tty_flip.h>
#include <linux/serial_core.h>
#include <linux/serial.h>
#include <asm/irq.h>
#include <asm/io.h>

#define USB_UART_MAJOR        200  /* You've to get this assigned */
#define USB_UART_MINOR_START  70   /* Start minor numbering here */
#define USB_UART_PORTS        2    /* The phone has 2 USB_UARTs */
#define PORT_USB_UART         30   /* UART type. Add this to
                                      include/linux/serial_core.h */

/* Each USB_UART has a 3-byte register set consisting of
   UU_STATUS_REGISTER at offset 0, UU_READ_DATA_REGISTER at
   offset 1, and UU_WRITE_DATA_REGISTER at offset 2 as shown
   in Table 6.1 */
#define USB_UART1_BASE    0xe8000000 /* Memory base for USB_UART1 */
#define USB_UART2_BASE    0xe9000000 /* Memory base for USB_UART2 */
#define USB_UART_REGISTER_SPACE  0x3

/* Semantics of bits in the status register */
#define USB_UART_TX_FULL         0x20  /* TX FIFO is full */
#define USB_UART_RX_EMPTY        0x10  /* TX FIFO is empty */
#define USB_UART_STATUS          0x0F  /* Parity/frame/overruns? */

#define USB_UART1_IRQ            3     /* USB_UART1 IRQ */
#define USB_UART2_IRQ            4     /* USB_UART2 IRQ */
#define USB_UART_FIFO_SIZE       32    /* FIFO size */
#define USB_UART_CLK_FREQ        16000000

static struct uart_port usb_uart_port[]; /* Defined later on */

/* Write a character to the USB_UART port */
static void
usb_uart_putc(struct uart_port *port, unsigned char c)
{
  /* Wait until there is space in the TX FIFO of the USB_UART.
     Sense this by looking at the USB_UART_TX_FULL bit in the
     status register */
  while (__raw_readb(port->membase) & USB_UART_TX_FULL);

  /* Write the character to the data port*/
  __raw_writeb(c, (port->membase+1));
}

/* Read a character from the USB_UART */
static unsigned char
usb_uart_getc(struct uart_port *port)
{
  /* Wait until data is available in the RX_FIFO */
  while (__raw_readb(port->membase) & USB_UART_RX_EMPTY);

  /* Obtain the data */
```

```
  return(__raw_readb(port->membase+2));
}

/* Obtain USB_UART status */
static unsigned char
usb_uart_status(struct uart_port *port)
{
  return(__raw_readb(port->membase) & USB_UART_STATUS);
}

/*
 * Claim the memory region attached to USB_UART port. Called
 * when the driver adds a USB_UART port via uart_add_one_port().
 */
static int
usb_uart_request_port(struct uart_port *port)
{
  if (!request_mem_region(port->mapbase, USB_UART_REGISTER_SPACE,
                          "usb_uart")) {
    return -EBUSY;
  }
  return 0;
}

/* Release the memory region attached to a USB_UART port.
 * Called when the driver removes a USB_UART port via
 * uart_remove_one_port().
 */
static void
usb_uart_release_port(struct uart_port *port)
{
  release_mem_region(port->mapbase, USB_UART_REGISTER_SPACE);
}

/*
 * Configure USB_UART. Called when the driver adds a USB_UART port.
 */
static void
usb_uart_config_port(struct uart_port *port, int flags)
{
  if (flags & UART_CONFIG_TYPE && usb_uart_request_port(port) == 0)
  {
    port->type = PORT_USB_UART;
  }
}

/* Receive interrupt handler */
static irqreturn_t
usb_uart_rxint(int irq, void *dev_id)
{
  struct uart_port *port = (struct uart_port *) dev_id;
  struct tty_struct *tty = port->info->tty;

  unsigned int status, data;
  /* ... */
  do {
    /* ... */
    /* Read data */
    data   = usb_uart_getc(port);
```

```
    /* Normal, overrun, parity, frame error? */
    status = usb_uart_status(port);
    /* Dispatch to the tty layer */
    tty_insert_flip_char(tty, data, status);
    /* ... */
  } while (more_chars_to_be_read()); /* More chars */
  /* ... */
  tty_flip_buffer_push(tty);

  return IRQ_HANDLED;
}
/* Called when an application opens a USB_UART */
static int
usb_uart_startup(struct uart_port *port)
{
  int retval = 0;
  /* ... */
  /* Request IRQ */
  if ((retval = request_irq(port->irq, usb_uart_rxint, 0,
                            "usb_uart", (void *)port))) {
    return retval;
  }
  /* ... */
  return retval;
}

/* Called when an application closes a USB_UART */
static void
usb_uart_shutdown(struct uart_port *port)
{
  /* ... */
  /* Free IRQ */
  free_irq(port->irq, port);

  /* Disable interrupts by writing to appropriate
     registers */
  /* ... */
}

/* Set UART type to USB_UART */
static const char *
usb_uart_type(struct uart_port *port)
{
  return port->type == PORT_USB_UART ? "USB_UART" : NULL;
}

/* Start transmitting bytes */
static void
usb_uart_start_tx(struct uart_port *port)
{
  while (1) {
    /* Get the data from the UART circular buffer and
       write it to the USB_UART's WRITE_DATA register */
    usb_uart_putc(port,
                  port->info->xmit.buf[port->info->xmit.tail]);
    /* Adjust the tail of the UART buffer */
    port->info->xmit.tail = (port->info->xmit.tail + 1) &
                            (UART_XMIT_SIZE - 1);
    /* Statistics */
```

```
    port->icount.tx++;
    /* Finish if no more data available in the UART buffer */
    if (uart_circ_empty(&port->info->xmit)) break;
  }
  /* ... */
}

/* The UART operations structure */
static struct uart_ops usb_uart_ops = {
  .start_tx     = usb_uart_start_tx,    /* Start transmitting */
  .startup      = usb_uart_startup,     /* App opens USB_UART */
  .shutdown     = usb_uart_shutdown,    /* App closes USB_UART */
  .type         = usb_uart_type,        /* Set UART type */
  .config_port  = usb_uart_config_port, /* Configure when driver
                                           adds a USB_UART port */
  .request_port = usb_uart_request_port,/* Claim resources
                                           associated with a
                                           USB_UART port */
  .release_port = usb_uart_release_port,/* Release resources
                                           associated with a
                                           USB_UART port */
#if 0    /* Left unimplemented for the USB_UART */
  .tx_empty     = usb_uart_tx_empty,    /* Transmitter busy? */
  .set_mctrl    = usb_uart_set_mctrl,   /* Set modem control */
  .get_mctrl    = usb_uart_get_mctrl,   /* Get modem control
  .stop_tx      = usb_uart_stop_tx,     /* Stop transmission */
  .stop_rx      = usb_uart_stop_rx,     /* Stop reception */
  .enable_ms    = usb_uart_enable_ms,   /* Enable modem status
                                           signals */
  .set_termios  = usb_uart_set_termios, /* Set termios */
#endif
};

static struct uart_driver usb_uart_reg = {
  .owner          = THIS_MODULE,          /* Owner */
  .driver_name    = "usb_uart",           /* Driver name */
  .dev_name       = "ttyUU",              /* Node name */
  .major          = USB_UART_MAJOR,       /* Major number */
  .minor          = USB_UART_MINOR_START, /* Minor number start */
  .nr             = USB_UART_PORTS,       /* Number of UART ports */
  .cons           = &usb_uart_console,    /* Pointer to the console
                                             structure. Discussed in Chapter
                                             12, "Video Drivers" */
};

/* Called when the platform driver is unregistered */
static int
usb_uart_remove(struct platform_device *dev)
{
  platform_set_drvdata(dev, NULL);

  /* Remove the USB_UART port from the serial core */
  uart_remove_one_port(&usb_uart_reg, &usb_uart_port[dev->id]);
  return 0;
}

/* Suspend power management event */
static int
usb_uart_suspend(struct platform_device *dev, pm_message_t state)
```

```
{
  uart_suspend_port(&usb_uart_reg, &usb_uart_port[dev->id]);
  return 0;
}

/* Resume after a previous suspend */
static int
usb_uart_resume(struct platform_device *dev)
{
  uart_resume_port(&usb_uart_reg, &usb_uart_port[dev->id]);
  return 0;
}

/* Parameters of each supported USB_UART port */
static struct uart_port usb_uart_port[] = {
  {
    .mapbase  = (unsigned int) USB_UART1_BASE,
    .iotype   = UPIO_MEM,           /* Memory mapped */
    .irq      = USB_UART1_IRQ,      /* IRQ */
    .uartclk  = USB_UART_CLK_FREQ,  /* Clock HZ */
    .fifosize = USB_UART_FIFO_SIZE, /* Size of the FIFO */
    .ops      = &usb_uart_ops,      /* UART operations */
    .flags    = UPF_BOOT_AUTOCONF,  /* UART port flag */
    .line     = 0,                  /* UART port number */
  },
  {
    .mapbase  = (unsigned int)USB_UART2_BASE,
    .iotype   = UPIO_MEM,           /* Memory mapped */
    .irq      = USB_UART2_IRQ,      /* IRQ */
    .uartclk  = USB_UART_CLK_FREQ,  /* CLock HZ */
    .fifosize = USB_UART_FIFO_SIZE, /* Size of the FIFO */
    .ops      = &usb_uart_ops,      /* UART operations */
    .flags    = UPF_BOOT_AUTOCONF,  /* UART port flag */
    .line     = 1,                  /* UART port number */
  }
};

/* Platform driver probe */
static int __init
usb_uart_probe(struct platform_device *dev)
{
  /* ... */

  /* Add a USB_UART port. This function also registers this device
     with the tty layer and triggers invocation of the config_port()
     entry point */
  uart_add_one_port(&usb_uart_reg, &usb_uart_port[dev->id]);
  platform_set_drvdata(dev, &usb_uart_port[dev->id]);
  return 0;
}

struct platform_device *usb_uart_plat_device1; /* Platform device
                                                  for USB_UART 1 */
struct platform_device *usb_uart_plat_device2; /* Platform device
                                                  for USB_UART 2 */

static struct platform_driver usb_uart_driver = {
  .probe   = usb_uart_probe,            /* Probe method */
  .remove  = __exit_p(usb_uart_remove), /* Detach method */
```

```
  .suspend = usb_uart_suspend,          /* Power suspend */
  .resume  = usb_uart_resume,           /* Resume after a suspend */
  .driver  = {
     .name = "usb_uart",                /* Driver name */
  },
};
/* Driver Initialization */
static int __init
usb_uart_init(void)
{
   int retval;

   /* Register the USB_UART driver with the serial core */
   if ((retval = uart_register_driver(&usb_uart_reg))) {
     return retval;
   }

   /* Register platform device for USB_UART 1. Usually called
      during architecture-specific setup */
   usb_uart_plat_device1 =
      platform_device_register_simple("usb_uart", 0, NULL, 0);
   if (IS_ERR(usb_uart_plat_device1)) {
     uart_unregister_driver(&usb_uart_reg);
     return PTR_ERR(usb_uart_plat_device1);
   }

   /* Register platform device for USB_UART 2. Usually called
      during architecture-specific setup */
   usb_uart_plat_device2 =
     platform_device_register_simple("usb_uart", 1, NULL, 0);
   if (IS_ERR(usb_uart_plat_device2)) {
     uart_unregister_driver(&usb_uart_reg);
     platform_device_unregister(usb_uart_plat_device1);
     return PTR_ERR(usb_uart_plat_device2);
   }

   /* Announce a matching driver for the platform
      devices registered above */
   if ((retval = platform_driver_register(&usb_uart_driver))) {
     uart_unregister_driver(&usb_uart_reg);
     platform_device_unregister(usb_uart_plat_device1);
     platform_device_unregister(usb_uart_plat_device2);
   }
   return 0;
}

/* Driver Exit */
static void __exit
usb_uart_exit(void)

{
  /* The order of unregistration is important. Unregistering the
     UART driver before the platform driver will crash the system */

  /* Unregister the platform driver */
  platform_driver_unregister(&usb_uart_driver);

  /* Unregister the platform devices */
  platform_device_unregister(usb_uart_plat_device1);
```

```
  platform_device_unregister(usb_uart_plat_device2);

  /* Unregister the USB_UART driver */
  uart_unregister_driver(&usb_uart_reg);
}

module_init(usb_uart_init);
module_exit(usb_uart_exit);
```

RS-485

RS-485 is not a standard PC interface like RS-232, but in the embedded space, you may come across computers that use RS-485 connections to reliably communicate with control systems. RS-485 uses differential signals that let it exchange data over distances of up to a few thousand feet, unlike RS-232 that has a range of only a few dozen feet. On the processor side, the RS-485 interface is a UART operating in half-duplex mode. So, before sending data from the transmit FIFO to the wire, the UART device driver needs to additionally enable the RS-485 transmitter and disable the receiver, possibly by wiggling associated GPIO pins. To obtain data from the wire to the receive FIFO, the UART driver has to perform the reverse operation.

You have to enable/disable the RS-485 transmitter/receiver at the right places in the serial layer. If you disable the transmitter too soon, it might not get sufficient time to drain the last bytes from the transmit FIFO, and this can result in data truncation. If you disable the transmitter too late, on the other hand, you prevent data reception for that much time, which might lead to receive data loss.

RS-485 supports multidrop, so the higher-layer protocol must implement a suitable addressing mechanism if you have multiple devices connected to the bus. RS-485 does not support hardware flow control lines using *Request To Send* (RTS) and *Clear To Send* (CTS).

TTY Drivers

Let's now take a look at the structures and registration functions that lie at the heart of tty drivers. Three structures are important for their operation:

1. `struct tty_struct` defined in *include/linux/tty.h*. This structure contains all state information associated with an open tty. It's an enormous structure, but here are some important fields:

```
struct tty_struct {
  int magic;                    /* Magic marker */
  struct tty_driver *driver;    /* Pointer to the tty
                                   driver */
  struct tty_ldisc ldisc;       /* Attached Line
                                   discipline */
  /* ... */
  struct tty_flip_buffer flip;  /* Flip Buffer. See
                                   below. */
  /* ... */

  wait_queue_head_t write_wait; /* See the section
                                   "Line Disciplines" */
  wait_queue_head_t read_wait;  /* See the section
                                   "Line Disciplines" */
  /* ... */
};
```

2. `struct tty_flip_buffer` or the *flip buffer* embedded inside `tty_struct`. This is the centerpiece of the data collection and processing mechanism:

```
struct tty_flip_buffer {
  /* ... */
  struct semaphore pty_sem;     /* Serialize */
  char *char_buf_ptr;           /* Pointer to the flip
                                   buffer */
  /* ... */
  unsigned char char_buf[2*TTY_FLIPBUF_SIZE]; /* The flip
                                                 buffer */
  /* ... */
};
```

   The low-level serial driver uses one half of the flip buffer for gathering data, while the line discipline uses the other half for processing the data. The buffer pointers used by the serial driver and the line discipline are then flipped, and the process continues. Have a look at the function `flush_to_ldisc()` in *drivers/char/tty_io.c* to see the flip in action.

   In recent kernels, the `tty_flip_buffer` structure has been somewhat redesigned. It's now made up of a buffer header (`tty_bufhead`) and a buffer list (`tty_buffer`):

```
struct tty_bufhead {
  /* ... */
  struct semaphore pty_sem;           /* Serialize */
```

```
  struct tty_buffer *head, tail, free; /* See below */
  /* ... */
};

struct tty_buffer {
  struct tty_buffer *next;
  char *char_buf_ptr;       /* Pointer to the flip buffer */
  /* ... */
  unsigned long data[0];    /* The flip buffer, memory for
                               which is dynamically
                               allocated */
};
```

3. `struct tty_driver` defined in *include/linux/tty_driver.h.* This specifies the programming interface between tty drivers and higher layers:

```
struct tty_driver {
  int magic;             /* Magic number */
  /* ... */
  int major;             /* Major number */
  int minor_start;       /* Start of minor number */
  /* ... */
  /* Interface routines between a tty driver and higher
     layers */
  int  (*open)(struct tty_struct *tty, struct file *filp);
  void (*close)(struct tty_struct *tty, struct file *filp);
  int  (*write)(struct tty_struct *tty,
                const unsigned char *buf, int count);
  void (*put_char)(struct tty_struct *tty,
                   unsigned char ch);
  /* ... */
};
```

Like a UART driver, a tty driver needs to perform two steps to register itself with the kernel:

1. Call `tty_register_driver(struct tty_driver *tty_d)` to register itself with the tty core.

2. Call

   ```
   tty_register_device(struct tty_driver *tty_d,
                       unsigned device_index,
                       struct device *device)
   ```

   to register each individual tty that it supports.

We won't develop a sample tty driver, but here are some common ones used on Linux:

- Serial emulation over Bluetooth, discussed in Chapter 16, is implemented in the form of a tty driver. This driver (*drivers/net/bluetooth/rfcomm/tty.c*) calls `tty_register_driver()` during initialization and `tty_register_device()` while handling each incoming Bluetooth connection.

- To work with a system console on a Linux desktop, you need the services of *virtual terminals* (VTs) if you

are in text mode or *pseudo terminals* (PTYs) if you are in graphics mode. VTs and PTYs are implemented as tty drivers and live in *drivers/char/vt.c* and *drivers/char/pty.c*, respectively.

- The tty driver used over conventional UARTs resides in *drivers/serial/serial_core.c*.

- The USB-serial tty driver is in *drivers/usb/serial/usb-serial.c*.

Line Disciplines

Line disciplines provide an elegant mechanism that lets you use the same serial driver to run different technologies. The low-level physical driver and the tty driver handle the transfer of data to and from the hardware, while line disciplines are responsible for processing the data and transferring it between kernel space and user space.

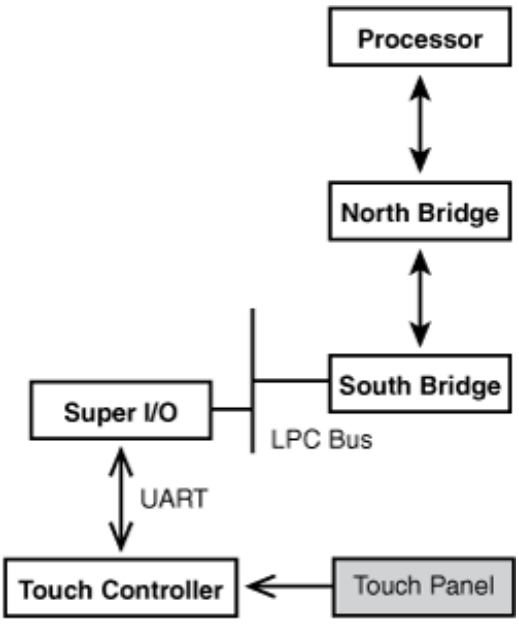
The serial subsystem supports 17 standard line disciplines. The default line discipline that gets attached when you open a serial port is `N_TTY`, which implements terminal I/O processing. `N_TTY` is responsible for "cooking" characters received from the keyboard. Depending on user request, it maps the control character to newline, converts lowercase to uppercase, expands tabs, and echoes characters to the associated VT. `N_TTY` also supports a *raw* mode used by editors, which leaves all the preceding processing to user applications. See Figure 7.3 in the next chapter, "Input Drivers," to learn how the keyboard subsystem is connected to `N_TTY`. The example tty drivers listed at the end of the previous section, "TTY Drivers," use `N_TTY` by default.

Line disciplines also implement network interfaces over serial transport protocols. For example, line disciplines that are part of the *Point-to-Point Protocol* (`N_PPP`) and the *Serial Line Internet Protocol* (`N_SLIP`) subsystems, frame packets, allocate and populate associated networking data structures, and pass the data on to the corresponding network protocol stack. Other line disciplines handle *Infrared Data* (`N_IRDA`) and the *Bluetooth Host Control Interface* (`N_HCI`).

Device Example: Touch Controller

Let's take a look at the internals of a line discipline by implementing a simple line discipline for a serial touch-screen controller. Figure 6.6 shows how the touch controller is connected on an embedded laptop derivative. The *Finite State Machine* (FSM) of the touch controller is a candidate for being implemented as a line discipline because it can leverage the facilities and interfaces offered by the serial layer.

Figure 6.6. Connection diagram of a touch controller on a PC-derivative.



Open and Close

To create a line discipline, define a `struct tty_ldisc` and register a prescribed set of entry points with the kernel. Listing 6.2 contains a code snippet that performs both these operations for the example touch controller.

Listing 6.2. Line Discipline Operations

```
Code View:
struct tty_ldisc n_touch_ldisc = {
  TTY_LDISC_MAGIC,         /* Magic */
  "n_tch",                 /* Name of the line discipline */
  N_TCH,                   /* Line discipline ID number */
  n_touch_open,            /* Open the line discipline */
  n_touch_close,           /* Close the line discipline */
  n_touch_flush_buffer,    /* Flush the line discipline's read
                              buffer */
  n_touch_chars_in_buffer, /* Get the number of processed characters in
                              the line discipline's read buffer */
  n_touch_read,            /* Called when data is requested
                              from user space */
  n_touch_write,           /* Write method */
  n_touch_ioctl,           /* I/O Control commands */
  NULL,                    /* We don't have a set_termios
                              routine */
  n_touch_poll,            /* Poll */
  n_touch_receive_buf,     /* Called by the low-level driver
                              to pass data to user space*/
  n_touch_receive_room,    /* Returns the room left in the line
                              discipline's read buffer */
  n_touch_write_wakeup     /* Called when the low-level device
                              driver is ready to transmit more
                              data */
};

/* ... */

if ((err = tty_register_ldisc(N_TCH, &n_touch_ldisc))) {
  return err;
}
```

In Listing 6.2, `n_tch` is the name of the line discipline, and `N_TCH` is the line discipline identifier number. You have to specify the value of `N_TCH` in *include/linux/tty.h*, the header file that contains all line discipline definitions. Line disciplines active on your system can be found in */proc/tty/ldiscs*.

Line disciplines gather data from their half of the tty flip buffer, process it, and copy the resulting data to a local *read* buffer. For `N_TCH`, `n_touch_receive_room()` returns the memory left in the read buffer, while `n_touch_chars_in_buffer()` returns the number of processed characters in the read buffer that are ready to be delivered to user space. `n_touch_write()` and `n_touch_write_wakeup()` do nothing because `N_TCH` is a read-only device. `n_touch_open()` takes care of allocating memory for the main line discipline data structures, as shown in Listing 6.3.

Listing 6.3. Opening the Line Discipline

Code View:

```
/* Private structure used to implement the Finite State Machine
(FSM) for the touch controller. The controller and the processor
communicate using a specific protocol that the FSM implements */
struct n_touch {
  int current_state;       /* Finite State Machine */
  spinlock_t touch_lock;   /* Spinlock */
  struct tty_struct *tty;  /* Associated tty */
  /* Statistics and other housekeeping */
  /* ... */
} *n_tch;


/* Device open() */
static int
n_touch_open(struct tty_struct *tty)
{
  /* Allocate memory for n_tch */
  if (!(n_tch = kmalloc(sizeof(struct n_touch), GFP_KERNEL))) {
    return -ENOMEM;
  }
  memset(n_tch, 0, sizeof(struct n_touch));

  tty->disc_data = n_tch; /* Other entry points now
                             have direct access to n_tch */
  /* Allocate the line discipline's local read buffer
     used for copying data out of the tty flip buffer */
  tty->read_buf = kmalloc(BUFFER_SIZE, GFP_KERNEL);
  if (!tty->read_buf) return -ENOMEM;

  /* Clear the read buffer */
  memset(tty->read_buf, 0, BUFFER_SIZE);

  /* Initialize lock */
  spin_lock_init(&ntch->touch_lock);

  /* Initialize other necessary tty fields.
     See drivers/char/n_tty.c for an example */
  /* ... */

  return 0;
}
```

You might want to set `N_TCH` as the default line discipline (rather than `N_TTY`) when-ever the serial port connected to the touch controller is opened. See the section "Changing Line Disciplines" to see how to change line disciplines from user space.

Read Path

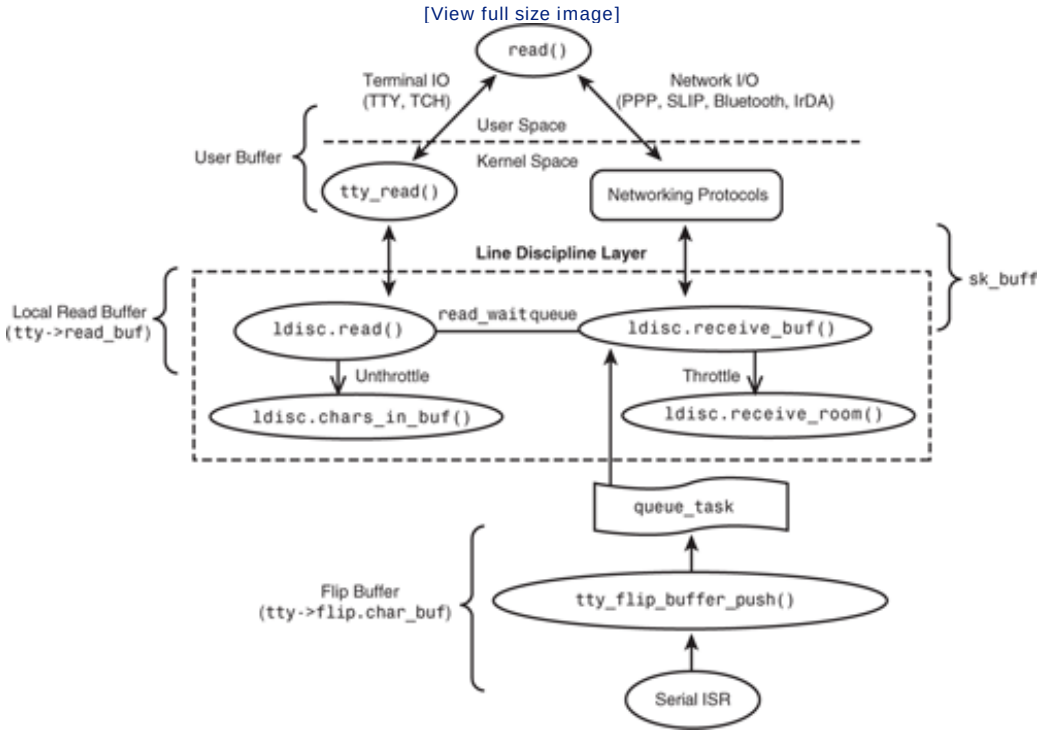
For interrupt-driven devices, the read data path usually consists of two threads working in tandem:

1. A top thread originating from the user process requesting the read

2. A bottom thread springing from the interrupt service routine that collects data from the device

Figure 6.7 shows these threads associated with the read data flow. The interrupt handler queues the `receive_buf()` method (`n_touch_receive_buf()` in our example) as a task. You can override this behavior by setting the `tty->low_latency` flag.

Figure 6.7. Line discipline read path.



The touch controller and the processor communicate using a specific protocol described in the controller's data sheet. The driver implements this communication protocol using an FSM as indicated earlier. Listing 6.4 encodes this FSM as part of the `receive_buf()` entry point, `n_touch_receive_buf()`.

Listing 6.4. The **`n_touch_receive_buf()`** Method

Code View:

```
static void
n_touch_receive_buf(struct tty_struct *tty,
                    const unsigned char *cp, char *fp, int count)
{

  /* Work on the data in the line discipline's half of
     the flip buffer pointed to by cp */
  /* ... */

  /* Implement the Finite State Machine to interpret commands/data
     arriving from the touch controller and put the processed data
     into the local read buffer */
.....................................................................................
  /* Datasheet-dependent Code Region */
  switch (tty->disc_data->current_state) {
    case RESET:
      /* Issue a reset command to the controller */
      tty->driver->write(tty, 0, mode_stream_command,
                         sizeof(mode_stream_command));
      tty->disc_data->current_state = STREAM_DATA;
      /* ... */
      break;
    case STREAM_DATA:
      /* ... */
      break;
    case PARSING:
      /* ... */
      tty->disc_data->current_state = PARSED;
      break;
    case PARSED:
      /* ... */
    }
.....................................................................................

  if (tty->disc_data->current_state == PARSED) {
    /* If you have a parsed packet, copy the collected coordinate
       and direction information into the local read buffer */
    spin_lock_irqsave(&tty->disc_data->touch_lock, flags);
    for (i=0; i < PACKET_SIZE; i++) {
      tty->disc_data->read_buf[tty->disc_data->read_head] =
                           tty->disc_data->current_pkt[i];
      tty->disc_data->read_head =
                 (tty->disc_data->read_head + 1) & (BUFFER_SIZE - 1);
      tty->disc_data->read_cnt++;
    }
    spin_lock_irqrestore(&tty->disc_data->touch_lock, flags);

    /* ... */ /* See Listing 6.5 */

  }
}
```

n_touch_receive_buf() processes the data arriving from the serial driver. It exchanges a series of commands and responses with the touch controller and puts the received coordinate and direction (press/release)

information into the line discipline's read buffer. Accesses to the read buffer have to be serialized using a spinlock because it's used by both `ldisc.receive_buf()` and `ldisc.read()` threads shown in Figure 6.7 (`n_touch_receive_buf()` and `n_touch_read()`, respectively, in our example). As you can see in Listing 6.4, `n_touch_receive_buf()` dispatches commands to the touch controller by directly calling the `write()` entry point of the serial driver.

`n_touch_receive_buf()` needs to do a couple more things:

1. The top `read()` thread in Figure 6.7 puts the calling process to sleep if no data is available. So, `n_touch_receive_buf()` has to wake up that thread and let it read the data that was just processed.

2. If the line discipline is running out of read buffer space, `n_touch_receive_buf()` has to request the serial driver to throttle data arriving from the device. `ldisc.read()` is responsible for requesting the corresponding unthrottling when it ferries the data to user space and frees memory in the read buffer. The serial driver uses software or hardware flow control mechanisms to achieve the throttling and unthrottling.

Listing 6.5 performs these two operations.

Listing 6.5. Waking Up the Read Thread and Throttling the Serial Driver

```
/* n_touch_receive_buf() continued.. */

/* Wake up any threads waiting for data */
if (waitqueue_active(&tty->read_wait) &&
    (tty->read_cnt >= tty->minimum_to_wake))
  wake_up_interruptible(&tty->read_wait);
}
/* If we are running out of buffer space, request the
   serial driver to throttle incoming data */
if (n_touch_receive_room(tty) < TOUCH_THROTTLE_THRESHOLD) {
  tty->driver.throttle(tty);
}
/* ... */
```

A wait queue (`tty->read_wait`) is used to synchronize between the `ldisc.read()` and `ldisc.receive_buf()` threads. `ldisc.read()` adds the calling process to the wait queue when it does not find data to read, while `ldisc.receive_buf()` wakes the `ldisc.read()` thread when there is data available to be read. So, `n_touch_read()` does the following:

- If there is no data to be read yet, put the calling process to sleep on the `read_wait` queue. The process gets woken by `n_touch_receive_buf()` when data arrives.

- If data is available, collect it from the local read buffer (`tty->read_buf[tty->read_tail]`) and dispatch it to user space.

- If the serial driver has been throttled and if enough space is available in the read buffer after this read, ask the serial driver to unthrottle.

Networking line disciplines usually allocate an `sk_buff` (the basic Linux networking data structure discussed in Chapter 15, "Network Interface Cards") and use this as the read buffer. They don't have a `read()` method, because the corresponding `receive_buf()` copies received data into the allocated `sk_buff` and directly passes it to the associated protocol stack.

Write Path

A line discipline's `write()` entry point performs any post processing that is required before passing the data down to the low-level driver.

If the underlying driver is not able to accept all the data that the line discipline offers, the line discipline puts the requesting thread to sleep. The driver's interrupt handler wakes the line discipline when it is ready to receive more data. To do this, the driver calls the `write_wakeup()` method registered by the line discipline. The associated synchronization is done using a wait queue (`tty->write_wait`), and the operation is similar to that of the `read_wait` queue described in the previous section.

Many networking line disciplines have no `write()` methods. The protocol implementation directly transmits the frames down to the serial device driver. However, these line disciplines usually still have a `write_wakeup()` entry point to respond to the serial driver's request for more transmit data.

`N_TCH` does not need a `write()` method either, because the touch controller is a read-only device. As you saw in Listing 6.4, routines in the receive path directly talk to the low-level UART driver when they need to send command frames to the controller.

I/O Control

A user program can send commands to a device via `ioctl()` calls, as discussed in Chapter 5, "Character Drivers." When an application opens a serial device, it can usually issue three classes of ioctls to it:

- Commands supported by the serial device driver, such as `TIOCMSET` that sets modem information
- Commands supported by the tty driver, such as `TIOCSETD` that changes the attached line discipline
- Commands supported by the attached line discipline, such as a command to reset the touch controller in the case of `N_TCH`

The `ioctl()` implementation for `N_TCH` is largely standard. Supported commands depend on the protocol described in the touch controller's data sheet.

More Operations

Another line discipline operation is `flush_buffer()`, which is used to flush any data pending in the read buffer. `flush_buffer()` is also called when a line discipline is closed. It wakes up any read threads that are waiting for more data as follows:

```
if (tty->link->packet){
  wake_up_interruptible(&tty->disc_data->read_wait);
}
```

Yet another entry point (not supported by `N_TCH`) is `set_termios()`. The `N_TTY` line discipline supports the

`set_termios()` interface, which is used to set options specific to line discipline data processing. For example, you may use `set_termios()` to put the line discipline in raw mode or "cooked" mode. Some options specific to the touch controller (such as changing the baud rate, parity, and number of stop bits) are implemented by the `set_termios()` method of the underlying device driver.

The remaining entry points such as `poll()` are fairly standard, and you can return to Chapter 5 in case you need assistance.

You may compile your line discipline as part of the kernel or dynamically load it as a module. If you choose to compile it as a module, you must, of course, also provide functions to be called during module initialization and exit. The former is usually the same as the `init()` method. The latter needs to clean up private data structures and unregister the line discipline. Unregistering the discipline is a one-liner:

```
tty_unregister_ldisc(N_TCH);
```

An easier way to drive a serial touch controller is by leveraging the services offered by the kernel's *input* subsystem and the built-in *serport* line discipline. We look at that technique in the next chapter.

Changing Line Disciplines

`N_TCH` gets bound to the low-level serial driver when a user space program opens the serial port connected to the touch controller. But sometimes, a user-space application might want to attach a different line discipline to the device. For instance, you might want to write a program that dumps raw data received from the touch controller without processing it. Listing 6.6 opens the touch controller and changes the line discipline to `N_TTY` to dump the data that is coming in.

Listing 6.6. Changing a Line Discipline from User Space

```
fd = open("/dev/ttySX", O_RDONLY | O_NOCTTY);

/* At this point, N_TCH is attached to /dev/ttySX, the serial port used
   by the touch controller. Switch to N_TTY */
ldisc = N_TTY;
ioctl(fd, TIOCSETD, &ldisc);

/* Set termios to raw mode and dump the data coming in */
/* ... */
```

The `TIOCSETD ioctl()` closes the current line discipline and opens the newly requested line discipline.

Looking at the Sources

The serial core resides in *drivers/serial/*, but tty implementations and low-level drivers are scattered across the source tree. The driver files referred to in Figure 6.3, for example, live in four different directories: *drivers/serial/, drivers/char/, drivers/usb/serial/*, and *drivers/net/irda/.* The *drivers/serial/* directory, which now also contains UART drivers, didn't exist in the 2.4 kernel; UART-specific code used to be dispersed between *drivers/char/* and *arch/your-arch/* directories. The present code partitioning is more logical because UART drivers are not the only folks that access the serial layer—devices such as USB-to-serial converters and IrDA dongles also need to talk to the serial core.

Look at *drivers/serial/imx.c* for a real-world, low-level UART driver. It handles UARTs that are part of Freescale's i.MX series of embedded controllers.

For a list of line disciplines supported on Linux, see *include/linux/tty.h.* To get a feel of networking line disciplines, look at the corresponding source files for PPP (*drivers/net/ppp_async.c*), Bluetooth (*drivers/bluetooth/hci_ldisc.c*), Infrared (*drivers/net/irda/irtty-sir.c*), and SLIP (*drivers/net/slip.c*).

Table 6.3 contains a summary of the main data structures used in this chapter and the location of their definitions in the source tree. Table 6.4 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 6.3. Summary of Data Structures

Data Structure	Location	Description
`uart_driver`	*include/linux/serial_core.h*	Representation of a low-level UART driver.
`uart_port`	*include/linux/serial_core.h*	Representation of a UART port.
`uart_ops`	*include/linux/serial_core.h*	Entry points supported by UART drivers.
`platform_device`	*include/linux/platform_device.h*	Representation of a platform device.
`platform_driver`	*include/linux/platform_device.h*	Representation of a platform driver.
`tty_struct`	*include/linux/tty.h*	State information about a tty.
`tty_bufhead`, `tty_buffer`	*include/linux/tty.h*	These two structures implement the flip buffer associated with a tty.
`tty_driver`	*include/linux/tty_driver.h*	Programming interface between tty drivers and higher layers.
`tty_ldisc`	*include/linux/tty_ldisc.h*	Entry points supported by a line discipline.

Table 6.4. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`uart_register_driver()`	*drivers/serial/sderial_core.c*	Registers a UART driver with the serial core
`uart_add_one_port()`	*drivers/serial/sderial_core.c*	Registers a UART port supported by the UART driver
`uart_unregister_driver()`	*drivers/serial/sderial_core.c*	Removes a UART driver from the serial core

`platform_device register()`
`platform_device_register_simple()`
`platform_add_devices()`
	drivers/base/platform.c	Registers a platform device

`platform_device_unregister()`	*drivers/base/platform.c*	Unregisters a platform device
`platform_driver_register()/` `platform_driver_unregister()`	*drivers/base/platform.c*	Registers/unregisters a platform driver
`tty_insert_flip_char()`	*include/linux/tty_flip.h*	Adds a character to the tty flip buffer
`tty_flip_buffer_push()`	*drivers/char/tty_io.c*	Queues a request to push the flip buffer to the line discipline
`tty_register_driver()`	*drivers/char/tty_io.c*	Registers a tty driver with the serial core
`tty_unregister_driver()`	*drivers/char/tty_io.c*	Removes a tty driver from the serial core
`tty_register_ldisc()`	*drivers/char/tty_io.c*	Creates a line discipline by registering prescribed entry points
`tty_unregister_ldisc()`	*drivers/char/tty_io.c*	Removes a line discipline from the serial core

Some serial data transfer scenarios are complex. You might need to mix and match different serial layer blocks, as you saw in Figure 6.3. Some situations may necessitate a data path passing through multiple line disciplines. For example, setting up a dialup connection over Bluetooth involves the movement of data through the HCI line discipline as well as the PPP line discipline. If you can, establish such a connection and step through the code flow using a kernel debugger.

Chapter 7. Input Drivers

In This Chapter

- Input Event Drivers 210
- Input Device Drivers 216
- Debugging 230
- Looking at the Sources 231

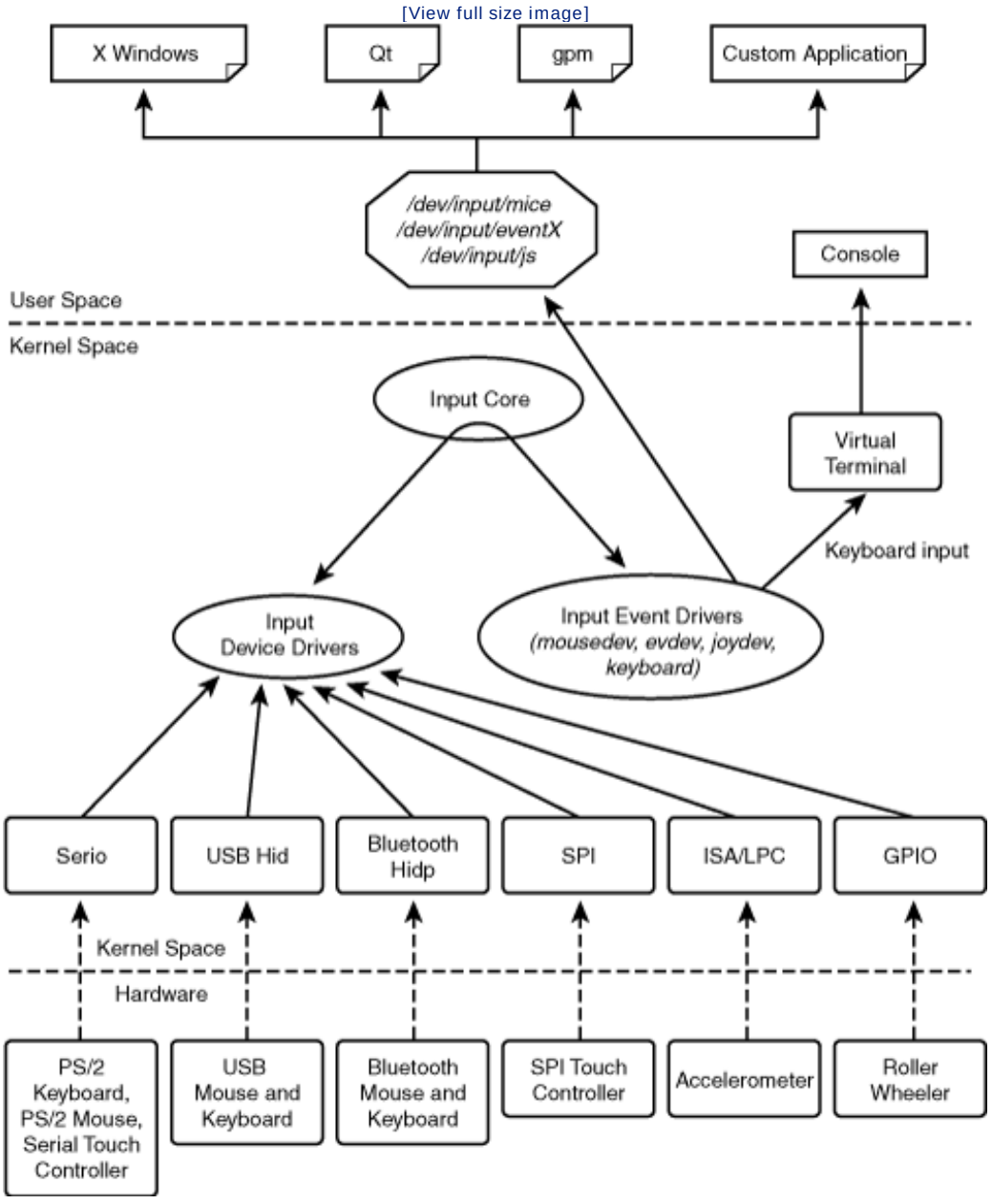
The kernel's *input* subsystem was created to unify scattered drivers that handle diverse classes of data-input devices such as keyboards, mice, trackballs, joysticks, roller wheels, touch screens, accelerometers, and tablets. The input subsystem brings the following advantages to the table:

- Uniform handling of functionally similar input devices even when they are physically different. For example, all mice, such as PS/2, USB or Bluetooth, are treated alike.
- An easy *event* interface for dispatching input reports to user applications. Your driver does not have to create and manage */dev* nodes and related access methods. Instead, it can simply invoke input APIs to send mouse movements, key presses, or touch events upstream to user land. Applications such as X Windows work seamlessly over the event interfaces exported by the input subsystem.
- Extraction of common portions out of input drivers and a resulting abstraction that simplifies the drivers and introduces consistency. For example, the input subsystem offers a collection of low-level drivers called *serio* that provides access to input hardware such as serial ports and keyboard controllers.

Figure 7.1 illustrates the operation of the input subsystem. The subsystem contains two classes of drivers that work in tandem: *event* drivers and *device* drivers. Event drivers are responsible for interfacing with applications, whereas device drivers are responsible for low-level communication with input devices. The mouse event generator, *mousedev*, is an example of the former, and the PS/2 mouse driver is an example of the latter. Both event drivers and device drivers can avail the services of an efficient, bug-free, reusable core, which lies at the heart of the input subsystem.

Figure 7.1. The input subsystem.

[View full size image]



Because event drivers are standardized and available for all input classes, you are more likely to implement a device driver than an event driver. Your device driver can use a suitable existing event driver via the input core to interface with user applications. Note that this chapter uses the term *device driver* to refer to an input device driver as opposed to an input event driver.

Input Event Drivers

The event interfaces exported by the input subsystem have evolved into a standard that many graphical windowing systems understand. Event drivers offer a hardware-independent abstraction to talk to input devices, just as the frame buffer interface (discussed in Chapter 12, "Video Drivers") presents a generic mechanism to communicate with display devices. Event drivers, in tandem with frame buffer drivers, insulate *graphical user interfaces* (GUIs) from the vagaries of the underlying hardware.

The Evdev Interface

Evdev is a generic input event driver. Each event packet produced by evdev has the following format, defined in *include/linux/input.h*:

```
struct input_event {
  struct timeval time;  /* Timestamp */
  __u16 type;           /* Event Type */
  __u16 code;           /* Event Code */
  __s32 value;          /* Event Value */
};
```

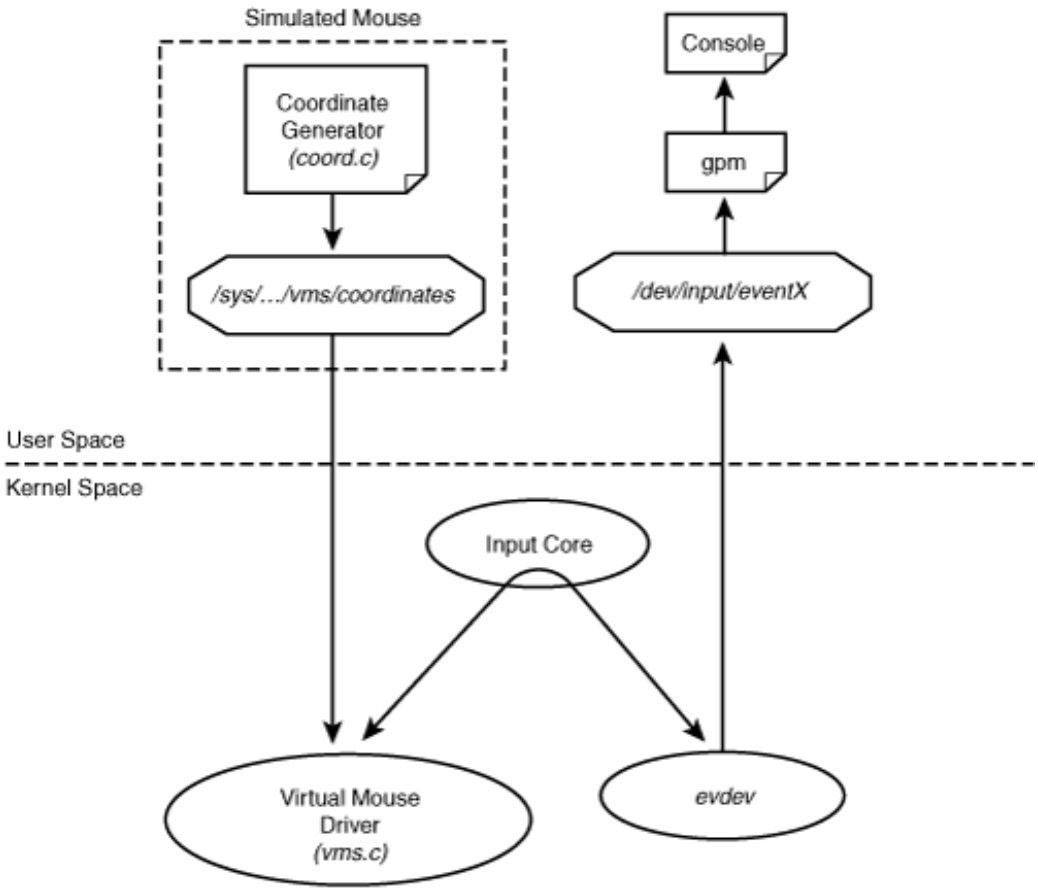
To learn how to use evdev, let's implement an input device driver for a virtual mouse.

Device Example: Virtual Mouse

This is how our virtual mouse works: An application (*coord.c*) emulates mouse movements and dispatches coordinate information to the virtual mouse driver (*vms.c*) via a sysfs node, */sys/devices/platform/vms/coordinates.* The virtual mouse driver (*vms* driver for short) channels these movements upstream via evdev. Figure 7.2 shows the details.

Figure 7.2. An input driver for a virtual mouse.

[View full size image]



General-purpose mouse (*gpm*) is a server that lets you use a mouse in text mode without assistance from an X server. Gpm understands evdev messages, so the vms driver can directly communicate with it. After you have everything in place, you can see the cursor dancing over your screen to the tune of the virtual mouse movements streamed by *coord.c*.

Listing 7.1 contains *coord.c*, which continuously generates random X and Y coordinates. Mice, unlike joysticks or touch screens, produce relative coordinates, so that is what *coord.c* does. The vms driver is shown in Listing 7.2.

Listing 7.1. Application to Simulate Mouse Movements (coord.c)

Code View:

```
#include <fcntl.h>

int
main(int argc, char *argv[])
{
  int sim_fd;
  int x, y;
  char buffer[10];

  /* Open the sysfs coordinate node */
  sim_fd = open("/sys/devices/platform/vms/coordinates", O_RDWR);
  if (sim_fd < 0) {
    perror("Couldn't open vms coordinate file\n");
    exit(-1);
  }
  while (1) {
    /* Generate random relative coordinates */
    x = random()%20;
    y = random()%20;
    if (x%2) x = -x; if (y%2) y = -y;

    /* Convey simulated coordinates to the virtual mouse driver */
    sprintf(buffer, "%d %d %d", x, y, 0);
    write(sim_fd, buffer, strlen(buffer));
    fsync(sim_fd);
    sleep(1);
  }

  close(sim_fd);
}
```

Listing 7.2. Input Driver for the Virtual Mouse (vms.c)

Code View:

```
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/pci.h>
#include <linux/input.h>
#include <linux/platform_device.h>

struct input_dev *vms_input_dev;        /* Representation of an input device */
static struct platform_device *vms_dev; /* Device structure */

                                        /* Sysfs method to input simulated
                                           coordinates to the virtual
                                           mouse driver */
static ssize_t
write_vms(struct device *dev,
          struct device_attribute *attr,
          const char *buffer, size_t count)
{
  int x,y;
  sscanf(buffer, "%d%d", &x, &y);
```

```
                                        /* Report relative coordinates via the
                                           event interface */
  input_report_rel(vms_input_dev, REL_X, x);
  input_report_rel(vms_input_dev, REL_Y, y);
  input_sync(vms_input_dev);

  return count;
}

/* Attach the sysfs write method */
DEVICE_ATTR(coordinates, 0644, NULL, write_vms);

/* Attribute Descriptor */
static struct attribute *vms_attrs[] = {
  &dev_attr_coordinates.attr,
  NULL
};

/* Attribute group */
static struct attribute_group vms_attr_group = {
  .attrs = vms_attrs,
};

/* Driver Initialization */
int __init
vms_init(void)
{

  /* Register a platform device */
  vms_dev = platform_device_register_simple("vms", -1, NULL, 0);
  if (IS_ERR(vms_dev)) {
    PTR_ERR(vms_dev);
    printk("vms_init: error\n");
  }

  /* Create a sysfs node to read simulated coordinates */
  sysfs_create_group(&vms_dev->dev.kobj, &vms_attr_group);

  /* Allocate an input device data structure */
  vms_input_dev = input_allocate_device();
  if (!vms_input_dev) {
    printk("Bad input_alloc_device()\n");
  }

  /* Announce that the virtual mouse will generate
     relative coordinates */
  set_bit(EV_REL, vms_input_dev->evbit);
  set_bit(REL_X, vms_input_dev->relbit);
  set_bit(REL_Y, vms_input_dev->relbit);

  /* Register with the input subsystem */
  input_register_device(vms_input_dev);

  printk("Virtual Mouse Driver Initialized.\n");
  return 0;
}

/* Driver Exit */
void
```

```
vms_cleanup(void)
{

  /* Unregister from the input subsystem */
  input_unregister_device(vms_input_dev);

  /* Cleanup sysfs node */
  sysfs_remove_group(&vms_dev->dev.kobj, &vms_attr_group);

  /* Unregister driver */
  platform_device_unregister(vms_dev);

  return;
}

module_init(vms_init);
module_exit(vms_cleanup);
```

Let's take a closer look at Listing 7.2. During initialization, the vms driver registers itself as an input device driver. For this, it first allocates an `input_dev` structure using the core API, `input_allocate_device()`:

```
vms_input_dev = input_allocate_device();
```

It then announces that the virtual mouse generates relative events:

```
set_bit(EV_REL, vms_input_dev->evbit);  /* Event Type is EV_REL */
```

Next, it declares the event codes that the virtual mouse produces:

```
set_bit(REL_X, vms_input_dev->relbit); /* Relative 'X' movement */
set_bit(REL_Y, vms_input_dev->relbit); /* Relative 'Y' movement */
```

If your virtual mouse is also capable of generating button clicks, you need to add this to `vms_init()`:

```
set_bit(EV_KEY, vms_input_dev->evbit);  /* Event Type is EV_KEY */
set_bit(BTN_0,  vms_input_dev->keybit); /* Event Code is BTN_0 */
```

Finally, the registration:

```
input_register_device(vms_input_dev);
```

`write_vms()` is the sysfs `store()` method that attaches to */sys/devices/platform/vms/coordinates.* When *coord.c* writes an X/Y pair to this file, `write_vms()` does the following:

```
input_report_rel(vms_input_dev, REL_X, x);
input_report_rel(vms_input_dev, REL_Y, y);
input_sync(vms_input_dev);
```

The first statement generates a `REL_X` event or a relative device movement in the X direction. The second produces a `REL_Y` event or a relative movement in the Y direction. `input_sync()` indicates that this event is complete, so the input subsystem collects these two events into a single evdev packet and sends it out of the door through */dev/input/eventX*, where *X* is the interface number assigned to the vms driver. An application reading this file will receive event packets in the `input_event` format described earlier. To request gpm to attach to this event interface and accordingly chase the cursor around your screen, do this:

```
bash> gpm -m /dev/input/eventX -t evdev
```

The ADS7846 touch controller driver and the accelerometer driver, discussed respectively under the sections "Touch Controllers" and "Accelerometers" later, are also evdev users.

More Event Interfaces

The vms driver utilizes the generic evdev event interface, but input devices such as keyboards, mice, and touch controllers have custom event drivers. We will look at them when we discuss the corresponding device drivers.

To write your own event driver and export it to user space via */dev/input/mydev*, you have to populate a structure called `input_handler` and register it with the input core as follows:

Code View:

```
static struct input_handler my_event_handler = {
  .event      = mydev_event,      /* Handle event reports sent by
                                     input device drivers that use
                                     this event driver's services */
  .fops       = &mydev_fops,      /* Methods to manage
                                     /dev/input/mydev */
  .minor      = MYDEV_MINOR_BASE, /* Minor number of
                                     /dev/input/mydev */
  .name       = "mydev",          /* Event driver name */
  .id_table   = mydev_ids,        /* This event driver can handle
                                     requests from these IDs */
  .connect    = mydev_connect,    /* Invoked if there is an
                                     ID match */
  .disconnect = mydev_disconnect, /* Called when the driver unregisters
                                      */
};

/* Driver Initialization */
static int __init
mydev_init(void)
{
  /* ... */

  input_register_handler(&my_event_handler);

  /* ... */
  return 0;
}
```

Look at the implementation of mousedev (*drivers/input/mousedev.c*) for a complete example.

Chapter 7. Input Drivers

In This Chapter

- Input Event Drivers 210
- Input Device Drivers 216
- Debugging 230
- Looking at the Sources 231

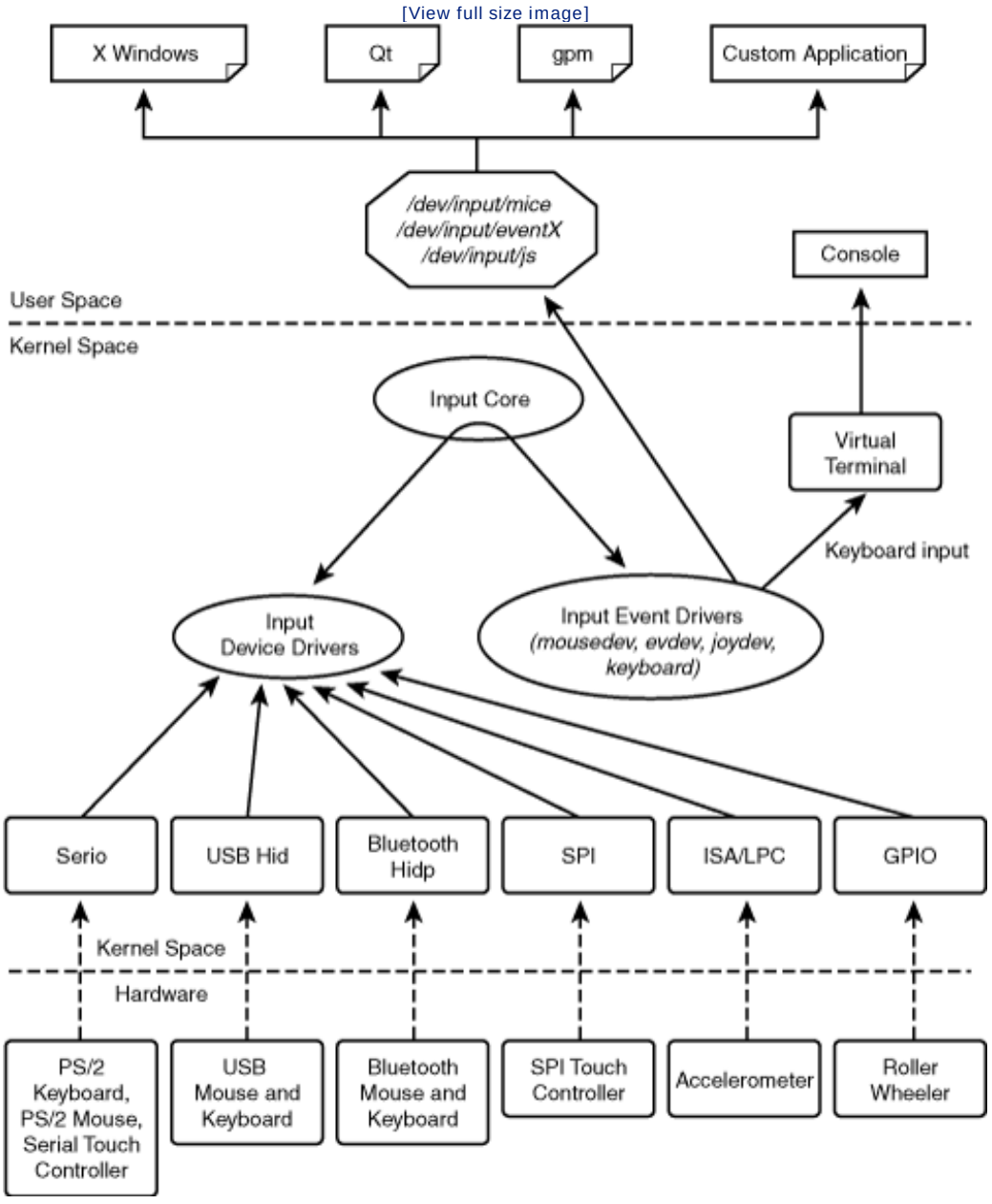
The kernel's *input* subsystem was created to unify scattered drivers that handle diverse classes of data-input devices such as keyboards, mice, trackballs, joysticks, roller wheels, touch screens, accelerometers, and tablets. The input subsystem brings the following advantages to the table:

- Uniform handling of functionally similar input devices even when they are physically different. For example, all mice, such as PS/2, USB or Bluetooth, are treated alike.
- An easy *event* interface for dispatching input reports to user applications. Your driver does not have to create and manage */dev* nodes and related access methods. Instead, it can simply invoke input APIs to send mouse movements, key presses, or touch events upstream to user land. Applications such as X Windows work seamlessly over the event interfaces exported by the input subsystem.
- Extraction of common portions out of input drivers and a resulting abstraction that simplifies the drivers and introduces consistency. For example, the input subsystem offers a collection of low-level drivers called *serio* that provides access to input hardware such as serial ports and keyboard controllers.

Figure 7.1 illustrates the operation of the input subsystem. The subsystem contains two classes of drivers that work in tandem: *event* drivers and *device* drivers. Event drivers are responsible for interfacing with applications, whereas device drivers are responsible for low-level communication with input devices. The mouse event generator, *mousedev*, is an example of the former, and the PS/2 mouse driver is an example of the latter. Both event drivers and device drivers can avail the services of an efficient, bug-free, reusable core, which lies at the heart of the input subsystem.

Figure 7.1. The input subsystem.

[View full size image]



Because event drivers are standardized and available for all input classes, you are more likely to implement a device driver than an event driver. Your device driver can use a suitable existing event driver via the input core to interface with user applications. Note that this chapter uses the term *device driver* to refer to an input device driver as opposed to an input event driver.

Input Event Drivers

The event interfaces exported by the input subsystem have evolved into a standard that many graphical windowing systems understand. Event drivers offer a hardware-independent abstraction to talk to input devices, just as the frame buffer interface (discussed in Chapter 12, "Video Drivers") presents a generic mechanism to communicate with display devices. Event drivers, in tandem with frame buffer drivers, insulate *graphical user interfaces* (GUIs) from the vagaries of the underlying hardware.

The Evdev Interface

Evdev is a generic input event driver. Each event packet produced by evdev has the following format, defined in *include/linux/input.h*:

```
struct input_event {
  struct timeval time;  /* Timestamp */
  __u16 type;           /* Event Type */
  __u16 code;           /* Event Code */
  __s32 value;          /* Event Value */
};
```

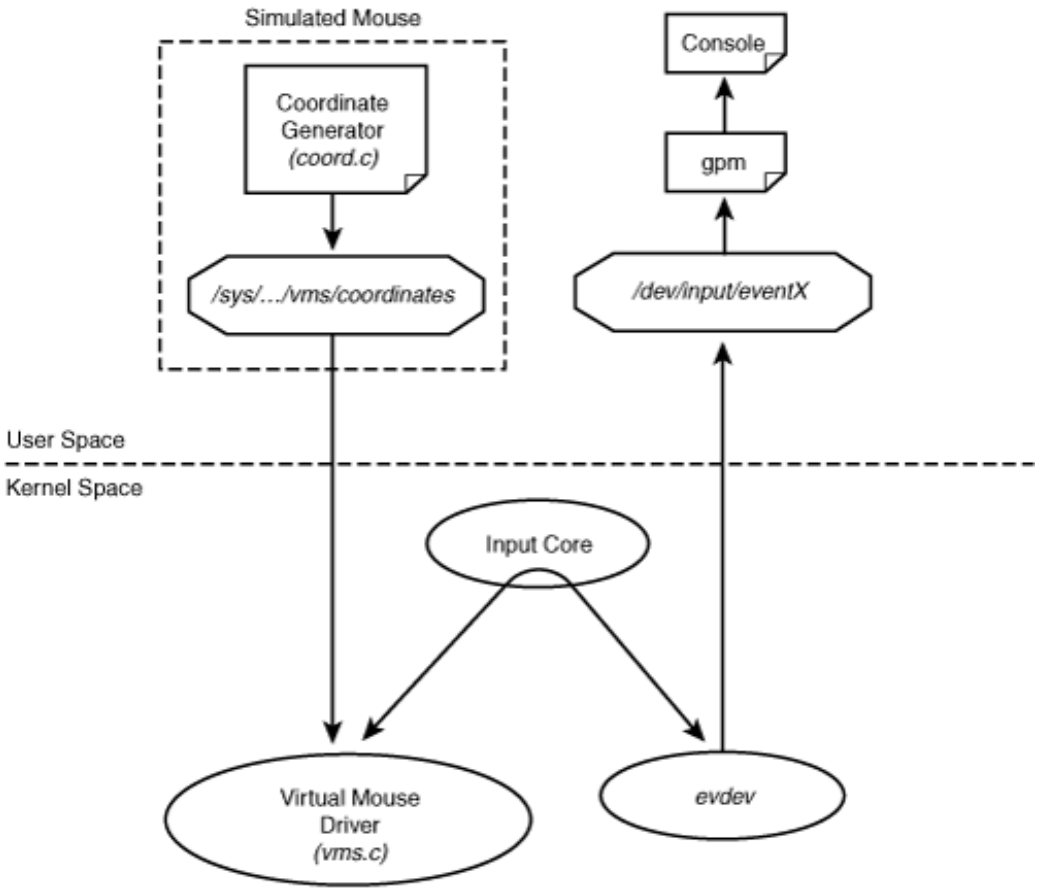
To learn how to use evdev, let's implement an input device driver for a virtual mouse.

Device Example: Virtual Mouse

This is how our virtual mouse works: An application (*coord.c*) emulates mouse movements and dispatches coordinate information to the virtual mouse driver (*vms.c*) via a sysfs node, */sys/devices/platform/vms/coordinates.* The virtual mouse driver (*vms* driver for short) channels these movements upstream via evdev. Figure 7.2 shows the details.

Figure 7.2. An input driver for a virtual mouse.

[View full size image]



General-purpose mouse (*gpm*) is a server that lets you use a mouse in text mode without assistance from an X server. Gpm understands evdev messages, so the vms driver can directly communicate with it. After you have everything in place, you can see the cursor dancing over your screen to the tune of the virtual mouse movements streamed by *coord.c*.

Listing 7.1 contains *coord.c*, which continuously generates random X and Y coordinates. Mice, unlike joysticks or touch screens, produce relative coordinates, so that is what *coord.c* does. The vms driver is shown in Listing 7.2.

Listing 7.1. Application to Simulate Mouse Movements (coord.c)

Code View:

```
#include <fcntl.h>

int
main(int argc, char *argv[])
{
  int sim_fd;
  int x, y;
  char buffer[10];

  /* Open the sysfs coordinate node */
  sim_fd = open("/sys/devices/platform/vms/coordinates", O_RDWR);
  if (sim_fd < 0) {
    perror("Couldn't open vms coordinate file\n");
    exit(-1);
  }
  while (1) {
    /* Generate random relative coordinates */
    x = random()%20;
    y = random()%20;
    if (x%2) x = -x; if (y%2) y = -y;

    /* Convey simulated coordinates to the virtual mouse driver */
    sprintf(buffer, "%d %d %d", x, y, 0);
    write(sim_fd, buffer, strlen(buffer));
    fsync(sim_fd);
    sleep(1);
  }

  close(sim_fd);
}
```

Listing 7.2. Input Driver for the Virtual Mouse (vms.c)

Code View:

```
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/pci.h>
#include <linux/input.h>
#include <linux/platform_device.h>

struct input_dev *vms_input_dev;        /* Representation of an input device */
static struct platform_device *vms_dev; /* Device structure */

                                        /* Sysfs method to input simulated
                                           coordinates to the virtual
                                           mouse driver */
static ssize_t
write_vms(struct device *dev,
          struct device_attribute *attr,
          const char *buffer, size_t count)
{
  int x,y;
  sscanf(buffer, "%d%d", &x, &y);
```

```
                                        /* Report relative coordinates via the
                                           event interface */
  input_report_rel(vms_input_dev, REL_X, x);
  input_report_rel(vms_input_dev, REL_Y, y);
  input_sync(vms_input_dev);

  return count;
}

/* Attach the sysfs write method */
DEVICE_ATTR(coordinates, 0644, NULL, write_vms);

/* Attribute Descriptor */
static struct attribute *vms_attrs[] = {
  &dev_attr_coordinates.attr,
  NULL
};

/* Attribute group */
static struct attribute_group vms_attr_group = {
  .attrs = vms_attrs,
};

/* Driver Initialization */
int __init
vms_init(void)
{

  /* Register a platform device */
  vms_dev = platform_device_register_simple("vms", -1, NULL, 0);
  if (IS_ERR(vms_dev)) {
    PTR_ERR(vms_dev);
    printk("vms_init: error\n");
  }

  /* Create a sysfs node to read simulated coordinates */
  sysfs_create_group(&vms_dev->dev.kobj, &vms_attr_group);

  /* Allocate an input device data structure */
  vms_input_dev = input_allocate_device();
  if (!vms_input_dev) {
    printk("Bad input_alloc_device()\n");
  }

  /* Announce that the virtual mouse will generate
     relative coordinates */
  set_bit(EV_REL, vms_input_dev->evbit);
  set_bit(REL_X, vms_input_dev->relbit);
  set_bit(REL_Y, vms_input_dev->relbit);

  /* Register with the input subsystem */
  input_register_device(vms_input_dev);

  printk("Virtual Mouse Driver Initialized.\n");
  return 0;
}

/* Driver Exit */
void
```

```
vms_cleanup(void)
{

  /* Unregister from the input subsystem */
  input_unregister_device(vms_input_dev);

  /* Cleanup sysfs node */
  sysfs_remove_group(&vms_dev->dev.kobj, &vms_attr_group);

  /* Unregister driver */
  platform_device_unregister(vms_dev);

  return;
}

module_init(vms_init);
module_exit(vms_cleanup);
```

Let's take a closer look at Listing 7.2. During initialization, the vms driver registers itself as an input device driver. For this, it first allocates an `input_dev` structure using the core API, `input_allocate_device()`:

```
vms_input_dev = input_allocate_device();
```

It then announces that the virtual mouse generates relative events:

```
set_bit(EV_REL, vms_input_dev->evbit);  /* Event Type is EV_REL */
```

Next, it declares the event codes that the virtual mouse produces:

```
set_bit(REL_X, vms_input_dev->relbit); /* Relative 'X' movement */
set_bit(REL_Y, vms_input_dev->relbit); /* Relative 'Y' movement */
```

If your virtual mouse is also capable of generating button clicks, you need to add this to `vms_init()`:

```
set_bit(EV_KEY, vms_input_dev->evbit);  /* Event Type is EV_KEY */
set_bit(BTN_0,  vms_input_dev->keybit); /* Event Code is BTN_0 */
```

Finally, the registration:

```
input_register_device(vms_input_dev);
```

`write_vms()` is the sysfs `store()` method that attaches to */sys/devices/platform/vms/coordinates.* When *coord.c* writes an X/Y pair to this file, `write_vms()` does the following:

```
input_report_rel(vms_input_dev, REL_X, x);
input_report_rel(vms_input_dev, REL_Y, y);
input_sync(vms_input_dev);
```

The first statement generates a `REL_X` event or a relative device movement in the X direction. The second produces a `REL_Y` event or a relative movement in the Y direction. `input_sync()` indicates that this event is complete, so the input subsystem collects these two events into a single evdev packet and sends it out of the door through */dev/input/eventX*, where *X* is the interface number assigned to the vms driver. An application reading this file will receive event packets in the `input_event` format described earlier. To request gpm to attach to this event interface and accordingly chase the cursor around your screen, do this:

```
bash> gpm -m /dev/input/eventX -t evdev
```

The ADS7846 touch controller driver and the accelerometer driver, discussed respectively under the sections "Touch Controllers" and "Accelerometers" later, are also evdev users.

More Event Interfaces

The vms driver utilizes the generic evdev event interface, but input devices such as keyboards, mice, and touch controllers have custom event drivers. We will look at them when we discuss the corresponding device drivers.

To write your own event driver and export it to user space via */dev/input/mydev*, you have to populate a structure called `input_handler` and register it with the input core as follows:

Code View:

```
static struct input_handler my_event_handler = {
  .event      = mydev_event,      /* Handle event reports sent by
                                     input device drivers that use
                                     this event driver's services */
  .fops       = &mydev_fops,      /* Methods to manage
                                     /dev/input/mydev */
  .minor      = MYDEV_MINOR_BASE, /* Minor number of
                                     /dev/input/mydev */
  .name       = "mydev",          /* Event driver name */
  .id_table   = mydev_ids,        /* This event driver can handle
                                     requests from these IDs */
  .connect    = mydev_connect,    /* Invoked if there is an
                                     ID match */
  .disconnect = mydev_disconnect, /* Called when the driver unregisters
                                      */
};

/* Driver Initialization */
static int __init
mydev_init(void)
{
  /* ... */

  input_register_handler(&my_event_handler);

  /* ... */
  return 0;
}
```

Look at the implementation of mousedev (*drivers/input/mousedev.c*) for a complete example.

Input Device Drivers

Let's turn our attention to drivers for common input devices such as keyboards, mice, and touch screens. But first, let's take a quick look at an off-the-shelf hardware access facility available to input drivers.

Serio

The *serio* layer offers library routines to access legacy input hardware such as i8042-compatible keyboard controllers and the serial port. PS/2 keyboards and mice interface with the former, whereas serial touch controllers connect to the latter. To communicate with hardware serviced by serio, for example, to send a command to a PS/2 mouse, register prescribed callback routines with serio using `serio_register_driver()`.

To add a new driver as part of serio, register `open()/close()/start()/stop()/write()` entry points using `serio_register_port ()`. Look at *drivers/input/serio/serport.c* for an example.

As you can see in Figure 7.1, serio is only one route to access low-level hardware. Several input device drivers instead rely on low-level support from bus layers such as USB or SPI.

Keyboards

Keyboards come in different flavors—legacy PS/2, USB, Bluetooth, Infrared, and so on. Each type has a specific input device driver, but all use the same keyboard event driver, thus ensuring a consistent interface to their users. The keyboard event driver, however, has a distinguishing feature compared to other event drivers: It passes data to another kernel subsystem (the tty layer), rather than to user space via */dev* nodes.

PC Keyboards

The PC keyboard (also called PS/2 keyboard or AT keyboard) interfaces with the processor via an i8042-compatible keyboard controller. Desktops usually have a dedicated keyboard controller, but on laptops, keyboard interfacing is one of the responsibilities of a general-purpose embedded controller (see the section "Embedded Controllers" in Chapter 20, "More Devices and Drivers"). When you press a key on a PC keyboard, this is the road it takes:

1. The keyboard controller (or the embedded controller) scans and decodes the keyboard matrix and takes care of nuances such as key debouncing.

2. The keyboard device driver, with the help of serio, reads raw *scancodes* from the keyboard controller for each key press and release. The difference between a press and a release is in the most significant bit, which is set for the latter. A push on the "a" key, for example, yields a pair of scancodes, `0x1e` and `0x9e`. Special keys are escaped using `0xE0`, so a jab on the right-arrow key produces the sequence, (`0xE0 0x4D 0xE0 0xCD`). You may use the *showkey* utility to observe scancodes emanating from the controller (the ⟶ symbol attaches explanations):

```
bash> showkey -s
kb mode was UNICODE
[ if you are trying this under X, it might not work since
 the X server is also reading /dev/console ]

 press any key (program terminates 10s after last
 keypress)...
```

```
...
0x1e 0x9e ⟶ A push of the "a" key
```

3. The keyboard device driver converts received scancodes to *keycodes*, based on the input mode. To see the keycode corresponding to the "a" key:

```
bash> showkey
...
keycode 30 press   ⟶ A push of the "a" key
keycode 30 release ⟶ Release of the "a" key
```

   To report the keycode upstream, the driver generates an input event, which passes control to the keyboard event driver.

4. The keyboard event driver undertakes keycode translation depending on the loaded key map. (See man pages of *loadkeys* and the map files present in */lib/kbd/keymaps.*) It checks whether the translated keycode is tied to actions such as switching the virtual console or rebooting the system. To glow the `CAPSLOCK` and `NUMLOCK` LEDs instead of rebooting the system in response to a Ctrl+Alt+Del push, add the following to the Ctrl+Alt+Del handler of the keyboard event driver, *drivers/char/keyboard.c*:

```
static void fn_boot_it(struct vc_data *vc,

                       struct pt_regs *regs)
{
+  set_vc_kbd_led(kbd, VC_CAPSLOCK);
+  set_vc_kbd_led(kbd, VC_NUMLOCK);
-  ctrl_alt_del();
}
```

5. For regular keys, the translated keycode is sent to the associated virtual terminal and the `N_TTY` line discipline. (We discussed virtual terminals and line disciplines in Chapter 6, "Serial Drivers.") This is done as follows by *drivers/char/keyboard.c*:

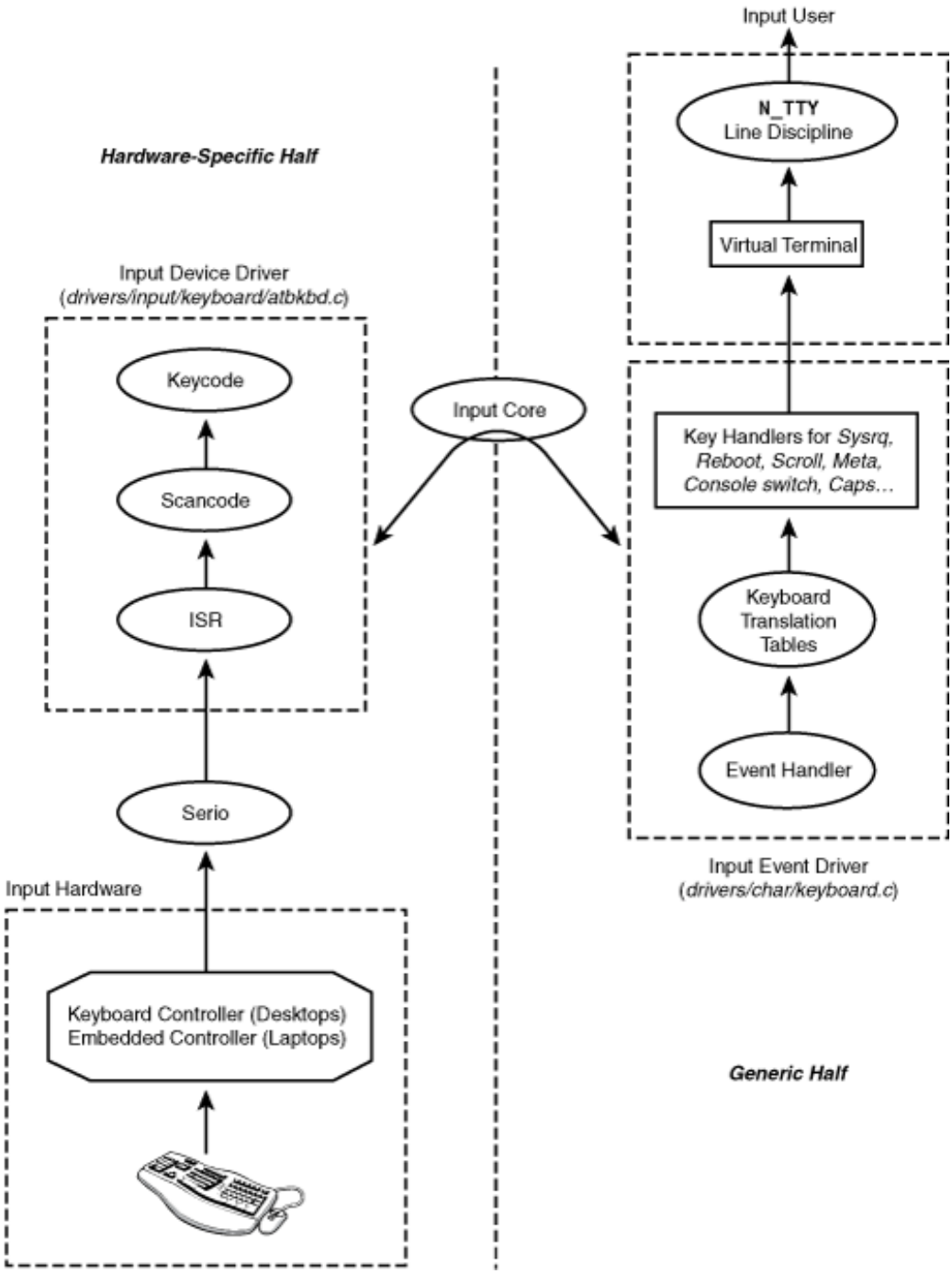
```
/* Add the keycode to flip buffer */
tty_insert_flip_char(tty, keycode, 0);
/* Schedule */
con_schedule_flip(tty);
```

The `N_TTY` line discipline processes the input thus received from the keyboard, echoes it to the virtual console, and lets user-space applications read characters from the */dev/ttyX* node connected to the virtual terminal.

Figure 7.3 shows the data flow from the time you push a key on your keyboard until the time it's echoed on your virtual console. The left half of the figure is hardware-specific, and the right half is generic. As per the design goal of the input subsystem, the underlying hardware interface is transparent to the keyboard event driver and the tty layer. The input core and the clearly defined event interfaces thus insulate input users from the intricacies of the hardware.

Figure 7.3. Data flow from a PS/2-compatible keyboard.

[View full size image]



USB and Bluetooth Keyboards

The USB specifications related to *human interface devices* (HID) stipulate the protocol that USB keyboards, mice, keypads, and other input peripherals use for communication. On Linux, this is implemented via the *usbhid* USB client driver, which is responsible for the USB HID class (`0x03`). Usbhid registers itself as an input device driver. It conforms to the input API and reports input events appropriate to the connected HID.

To understand the code path for a USB keyboard, revert to Figure 7.3 and modify the hardware-specific left half. Replace the keyboard controller in the Input Hardware box with a USB controller, serio with the USB core layer, and the Input Device Driver box with the usbhid driver.

For a Bluetooth keyboard, replace the keyboard controller in Figure 7.3 with a Bluetooth chipset, serio with the Bluetooth core layer, and the Input Device Driver box with the Bluetooth hidp driver.

USB and Bluetooth are discussed in detail in Chapter 11, "Universal Serial Bus," and Chapter 16, "Linux Without Wires," respectively.

Mice

Mice, like keyboards, come with different capabilities and have different interfacing options. Let's look at the common ones.

PS/2 Mice

Mice generate relative movements in the X and Y axes. They also possess one or more buttons. Some have scroll wheels, too. The input device driver for PS/2-compatible legacy mice relies on the serio layer to talk to the underlying controller. The input event driver for mice, called *mousedev*, reports mouse events to user applications via */dev/input/mice*.

Device Example: Roller Mouse

To get a feel of a real-world mouse device driver, let's convert the roller wheel discussed in Chapter 4, "Laying the Groundwork," into a variation of the generic PS/2 mouse. The "roller mouse" generates one-dimensional movement in the Y-axis. Clockwise and anticlockwise turns of the wheel produce positive and negative relative Y coordinates respectively (like the scroll wheel in mice), while pressing the roller wheel results in a left button mouse event. The roller mouse is thus ideal for navigating menus in devices such as smart phones, handhelds, and music players.

The roller mouse device driver implemented in Listing 7.3 works with windowing systems such as X Windows. Look at `roller_mouse_init()` to see how the driver declares its mouse-like capabilities. Unlike the roller wheel driver in Listing 4.1 of Chapter 4, the roller mouse driver needs no `read()` or `poll()` methods because events are reported using input APIs. The roller interrupt handler `roller_isr()` also changes accordingly. Gone are the housekeepings done in the interrupt handler using a wait queue, a spinlock, and the `store_movement()` routine to support `read()` and `poll()`.

In Listing 7.3, the leading **+** and **-** denote the differences from the roller wheel driver implemented in Listing 4.1 of Chapter 4.

Listing 7.3. The Roller Mouse Driver

Code View:

```
+  #include <linux/input.h>
+  #include <linux/interrupt.h>

+  /* Device structure */
+  struct {
+    /* ... */
+    struct input_dev dev;
+  } roller_mouse;

+  static int __init
+  roller_mouse_init(void)
+  {
+  /* Allocate input device structure */
+  roller_mouse->dev = input_allocate_device();
+
```

```
+  /* Can generate a click and a relative movement */
+  roller_mouse->dev->evbit[0] = BIT(EV_KEY) | BIT(EV_REL);

+  /* Can move only in the Y-axis */
+  roller_mouse->dev->relbit[0] = BIT(REL_Y);
+
+  /* My click should be construed as the left button
+     press of a mouse */
+  roller_mouse->dev->keybit[LONG(BTN_MOUSE)] = BIT(BTN_LEFT);

+  roller_mouse->dev->name = "roll";
+
+  /* For entries in /sys/class/input/inputX/id/ */
+  roller_mouse->dev->id.bustype = ROLLER_BUS;
+  roller_mouse->dev->id.vendor = ROLLER_VENDOR;
+  roller_mouse->dev->id.product = ROLLER_PROD;
+  roller_mouse->dev->id.version = ROLLER_VER;

+  /* Register with the input subsystem */
+  input_register_device(roller_mouse->dev);
+}

/* Global variables */
- spinlock_t roller_lock = SPIN_LOCK_UNLOCKED;
- static DECLARE_WAIT_QUEUE_HEAD(roller_poll);

/* The Roller Interrupt Handler */
static irqreturn_t
roller_interrupt(int irq, void *dev_id)
{
  int i, PA_t, PA_delta_t, movement = 0;

  /* Get the waveforms from bits 0, 1 and 2
     of Port D as shown in Figure 7.1 */
  PA_t = PORTD & 0x07;

  /* Wait until the state of the pins change.
     (Add some timeout to the loop) */
  for (i=0; (PA_t==PA_delta_t); i++){
     PA_delta_t = PORTD & 0x07;
  }

  movement = determine_movement(PA_t, PA_delta_t);

- spin_lock(&roller_lock);
-
- /* Store the wheel movement in a buffer for
-    later access by the read()/poll() entry points */
- store_movements(movement);
-
- spin_unlock(&roller_lock);
-
- /* Wake up the poll entry point that might have
-    gone to sleep, waiting for a wheel movement */
- wake_up_interruptible(&roller_poll);
-
+ if (movement == CLOCKWISE) {
+   input_report_rel(roller_mouse->dev, REL_Y, 1);
+ } else if (movement == ANTICLOCKWISE) {
```

```
+   input_report_rel(roller_mouse->dev, REL_Y, -1);
+ } else if (movement == KEYPRESSED) {
+   input_report_key(roller_mouse->dev, BTN_LEFT, 1);
+ }
+ input_sync(roller_mouse->dev);

  return IRQ_HANDLED;
}
```

Trackpoints

A *trackpoint* is a pointing device that comes integrated with the PS/2-type keyboard on several laptops. This device includes a joystick located among the keys and mouse buttons positioned under the spacebar. A trackpoint essentially functions as a mouse, so you can operate it using the PS/2 mouse driver.

Unlike a regular mouse, a trackpoint offers more movement control. You can command the trackpoint controller to change properties such as sensitivity and inertia. The kernel has a special driver, *drivers/input/mouse/trackpoint.c*, to create and manage associated sysfs nodes. For the full set of track point configuration options, look under */sys/devices/platform/i8042/serioX/serioY/*.

Touchpads

A *touchpad* is a mouse-like pointing device commonly found on laptops. Unlike conventional mice, a touchpad does not have moving parts. It can generate mouse-compatible relative coordinates but is usually used by operating systems in a more powerful mode that produces absolute coordinates. The communication protocol used in absolute mode is similar to the PS/2 mouse protocol, but not compatible with it.

The basic PS/2 mouse driver is capable of supporting devices that conform to different variations of the bare PS/2 mouse protocol. You may add support for a new mouse protocol to the base driver by supplying a protocol driver via the `psmouse` structure. If your laptop uses the Synaptics touchpad in absolute mode, for example, the base PS/2 mouse driver uses the services of a Synaptics protocol driver to interpret the streaming data. For an end-to-end understanding of how the Synaptics protocol works in tandem with the base PS/2 driver, look at the following four code regions collected in Listing 7.4:

- The PS/2 mouse driver, *drivers/input/mouse/psmouse-base.c*, instantiates a `psmouse_protocol` structure with information regarding supported mouse protocols (including the Synaptics touchpad protocol).
- The `psmouse` structure, defined in *drivers/input/mouse/psmouse.h*, ties various PS/2 protocols together.
- `synaptics_init()` populates the `psmouse` structure with the address of associated protocol functions.
- The protocol handler function `synaptics_process_byte()`, populated in `synaptics_init()`, gets called from interrupt context when serio senses mouse movement. If you unfold `synaptics_process_byte()`, you will see touchpad movements being reported to user applications via mousedev.

Listing 7.4. PS/2 Mouse Protocol Driver for the Synaptics Touchpad

Code View:

***drivers/input/mouse/psmouse-base.c*:**

```
/* List of supported PS/2 mouse protocols */
static struct psmouse_protocol psmouse_protocols[] = {
 {
   .type     = PSMOUSE_PS2, /* The bare PS/2 handler */
   .name     = "PS/2",
   .alias    = "bare",
   .maxproto = 1,
   .detect   = ps2bare_detect,
 },
 /*  ... */
 {
   .type    = PSMOUSE_SYNAPTICS, /* Synaptics TouchPad Protocol */
   .name    = "SynPS/2",
   .alias   = "synaptics",
   .detect  = synaptics_detect,  /* Is the protocol detected? */
   .init    = synaptics_init,    /* Initialize Protocol Handler */
 },
  /*  ... */
}
```

***drivers/input/mouse/psmouse.h*:**

```
/* The structure that ties various mouse protocols together */
struct psmouse {
  struct input_dev *dev; /* The input device */
  /* ... */

  /* Protocol Methods */
  psmouse_ret_t (*protocol_handler)
                (struct psmouse *psmouse, struct pt_regs *regs);
  void (*set_rate)(struct psmouse *psmouse, unsigned int rate);
  void (*set_resolution)
        (struct psmouse *psmouse, unsigned int resolution);
  int (*reconnect)(struct psmouse *psmouse);
  void (*disconnect)(struct psmouse *psmouse);
  /* ... */
};
```

***drivers/input/mouse/synaptics.c*:**

```
/* init() method of the Synaptics protocol */
int synaptics_init(struct psmouse *psmouse)
{
  struct synaptics_data *priv;
  psmouse->private = priv = kmalloc(sizeof(struct synaptics_data),
                                    GFP_KERNEL);
  /* ... */

  /* This is called in interrupt context when mouse
     movement is sensed */
  psmouse->protocol_handler = synaptics_process_byte;

  /* More protocol methods */
  psmouse->set_rate = synaptics_set_rate;
  psmouse->disconnect = synaptics_disconnect;
  psmouse->reconnect = synaptics_reconnect;

  /* ... */
```

```
}

drivers/input/mouse/synaptics.c:
/* If you unfold synaptics_process_byte() and look at
   synaptics_process_packet(), you can see the input
   events being reported to user applications via mousedev */
static void synaptics_process_packet(struct psmouse *psmouse)
{
  /* ... */
  if (hw.z > 0) {
    /* Absolute X Coordinate */
    input_report_abs(dev, ABS_X, hw.x);
    /* Absolute Y Coordinate */
    input_report_abs(dev, ABS_Y,
                     YMAX_NOMINAL + YMIN_NOMINAL - hw.y);
  }
  /* Absolute Z Coordinate */
  input_report_abs(dev, ABS_PRESSURE, hw.z);
  /* ... */
  /* Left TouchPad button */
  input_report_key(dev, BTN_LEFT, hw.left);
  /* Right TouchPad button */
  input_report_key(dev, BTN_RIGHT, hw.right);
  /* ... */
}
```

USB and Bluetooth Mice

USB mice are handled by the same input driver (*usbhid*) that drives USB keyboards. Similarly, the *hidp* driver that implements support for Bluetooth keyboards also takes care of Bluetooth mice.

As you would expect, USB and Bluetooth mice drivers channel device data through mousedev.

Touch Controllers

In Chapter 6, we implemented a device driver for a serial touch controller in the form of a line discipline called `N_TCH`. The input subsystem offers a better and easier way to implement that driver. Refashion the finite state machine in `N_TCH` as an input device driver with the following changes:

1. Serio offers a line discipline called *serport* for accessing devices connected to the serial port. Use serport's services to talk to the touch controller.

2. Instead of passing coordinate information to the tty layer, generate input reports via evdev as you did in Listing 7.2 for the virtual mouse.

With this, the touch screen is accessible to user space via */dev/input/eventX.* The actual driver implementation is left as an exercise.

An example of a touch controller that does not interface via the serial port is the Analog Devices ADS7846 chip, which communicates over a *Serial Peripheral Interface* (SPI). The driver for this device uses the services of the SPI core rather than serio. The section "The Serial Peripheral Interface Bus" in Chapter 8, "The Inter-Integrated Circuit Protocol," discusses SPI. Like most touch drivers, the ADS7846 driver uses the evdev interface to dispatch touch information to user applications.

Some touch controllers interface over USB. An example is the 3M USB touch controller, driven by *drivers/input/touchscreen/usbtouchscreen.c*.

> Many PDAs have four-wire resistive touch panels superimposed on their LCDs. The X and Y plates of the panel (two wires for either axes) connect to an analog-to-digital converter (ADC), which provides a digital readout of the analog voltage difference arising out of touching the screen. An input driver collects the coordinates from the ADC and dispatches it to user space.

Different instances of the same touch panel may produce slightly different coordinate ranges (maximum values in the X and Y directions) due to the nuances of manufacturing processes. To insulate applications from this variation, touch screens are *calibrated* prior to use. Calibration is usually initiated by the GUI by displaying cross-marks at screen boundaries and other vantage points, and requesting the user to touch those points. The generated coordinates are programmed back into the touch controller using appropriate commands if it supports self-calibration, or used to scale the coordinate stream in software otherwise.

The input subsystem also contains an event driver called *tsdev* that generates coordinate information according to the Compaq touch-screen protocol. If your system reports touch events via *tsdev*, applications that understand this protocol can elicit touch input from */dev/input/tsX.* This driver is, however, scheduled for removal from the mainline kernel in favor of the user space *tslib* library. *Documentation/feature-removal-schedule.txt* lists features that are going away from the kernel source tree.

Accelerometers

An accelerometer measures acceleration. Several IBM/Lenovo laptops have an accelerometer that detects sudden movement. The generated information is used to protect the hard disk from damage using a mechanism called *Hard Drive Active Protection System* (HDAPS), analogous to the way a car airbag shields a passenger from injury. The HDAPS driver is implemented as a platform driver that registers with the input subsystem. It uses evdev to stream the X and Y components of the detected acceleration. Applications can read acceleration events via */dev/input/eventX* to detect conditions, such as shock and vibe, and perform a defensive action, such as parking the hard drive's head. The following command spews output if you move the laptop (assume that event3 is assigned to HDAPS):

```
bash> od –x /dev/input/event3
0000000 a94d 4599 1f19 0007 0003 0000 ffed ffff
...
```

The accelerometer also provides information such as temperature, keyboard activity, and mouse activity, all of which can be gleaned via files in */sys/devices/platform/hdaps/.* Because of this, the HDAPS driver is part of the hardware monitoring (*hwmon*) subsystem in the kernel sources. We talk about hardware monitoring in the section "Hardware Monitoring with LM-Sensors" in the next chapter.

Output Events

Some input device drivers also handle output events. For example, the keyboard driver can glow the CAPSLOCK

LED, and the PC speaker driver can sound a beep. Let's zoom in on the latter. During initialization, the speaker driver declares its output capability by setting appropriate *evbits* and registering a callback routine to handle the output event:

Code View:
***drivers/input/misc/pcspkr.c*:**

```
static int __devinit pcspkr_probe(struct platform_device *dev)
{
  /* ... */

  /* Capability Bits */
  pcspkr_dev->evbit[0]  = BIT(EV_SND);
  pcspkr_dev->sndbit[0] = BIT(SND_BELL) | BIT(SND_TONE);

  /* The Callback routine */
  pcspkr_dev->event = pcspkr_event;

  err = input_register_device(pcspkr_dev);
  /* ... */
}

/* The callback routine */
static int pcspkr_event(struct input_dev *dev, unsigned int type,
                        unsigned int code, int value)
{

  /* ... */

  /* I/O programming to sound a beep */

  outb_p(inb_p(0x61) | 3, 0x61);
  /* set command for counter 2, 2 byte write */
  outb_p(0xB6, 0x43);
  /* select desired HZ */
  outb_p(count & 0xff, 0x42);
  outb((count >> 8) & 0xff, 0x42);

  /* ... */
}
```

To sound the beeper, the keyboard event driver generates a sound event (`EV_SND`) as follows:

```
input_event(handle->dev, EV_SND,    /* Type */
                         SND_TONE,  /* Code */
                         hz         /* Value */);
```

This triggers execution of the callback routine, `pcspkr_event()`, and you hear the beep.

Debugging

You can use the *evbug* module as a debugging aid if you're developing an input driver. It dumps the (*type*, *code*, *value*) tuple (see `struct input_event` defined previously) corresponding to events generated by the input subsystem. Figure 7.4 contains data captured by evbug while operating some input devices:

Figure 7.4. Evbug output.

Code View:

```
/* Touchpad Movement */
evbug.c Event. Dev: isa0060/serio1/input0: Type: 3, Code: 28, Value: 0
evbug.c Event. Dev: isa0060/serio1/input0: Type: 1, Code: 325, Value: 0
evbug.c Event. Dev: isa0060/serio1/input0: Type: 0, Code: 0, Value: 0

/* Trackpoint Movement */
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 0, Value: -1
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 1, Value: -2
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 0, Code: 0, Value: 0

/* USB Mouse Movement */
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 1, Value: -1
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 0, Value: 1
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0

/* PS/2 Keyboard keypress 'a' */
evbug.c Event. Dev: isa0060/serio0/input0: Type: 4, Code: 4, Value: 30
evbug.c Event. Dev: isa0060/serio0/input0: Type: 1, Code: 30, Value: 0
evbug.c Event. Dev: isa0060/serio0/input0: Type: 0, Code: 0, Value: 0

/* USB keyboard keypress 'a' */
evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 1, Code: 30, Value: 1
evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 0, Code: 0, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 1, Code: 30, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0
```

To make sense of the dump in Figure 7.4, remember that touchpads generate absolute coordinates (`EV_ABS`) or event type `0x03`, trackpoints produce relative coordinates (`EV_REL`) or event type `0x02`, and keyboards emit key events (`EV_KEY`) or event type `0x01`. Event type `0x0` corresponds to an invocation of `input_sync()`, which does the following:

```
input_event(dev, EV_SYN, SYN_REPORT, 0);
```

This translates to a (*type, code, value*) tuple of (`0x0`, `0x0`, `0x0`) and completes each input event.

Looking at the Sources

Most input event drivers are present in the *drivers/input/* directory. The keyboard event driver, however, lives in *drivers/char/keyboard.c*, because it's connected to virtual terminals and not to device nodes under */dev/input/*.

You can find input device drivers in several places. Drivers for legacy keyboards, mice, and joysticks, reside in separate subdirectories under *drivers/input/.* Bluetooth input drivers live in *net/bluetooth/hidp/.* You can also find input drivers in regions such as *drivers/hwmon/* and *drivers/media/video/.* Event types, codes, and values, are defined in *include/linux/input.h*.

The serio subsystem stays in *drivers/input/serio/.* Sources for the serport line discipline is in *drivers/input/serio/serport.c. Documentation/input/* contains more details on different input interfaces.

Table 7.1 summarizes the main data structures used in this chapter and their location inside the source tree. Table 7.2 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 7.1. Summary of Data Structures

Data Structure	Location	Description
`input_event`	*include/linux/input.h*	Each event packet produced by evdev has this format.
`input_dev`	*include/linux/input.h*	Representation of an input device.
`input_handler`	*include/linux/serial_core.h*	Contains the entry points supported by an event driver.
`psmouse_protocol`	*drivers/input/mouse/psmouse-base.c*	Information about a supported PS/2 mouse protocol driver.
`psmouse`	*drivers/input/mouse/psmouse.h*	Methods supported by a PS/2 mouse driver.

Table 7.2. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`input_register_device()`	*drivers/input/input.c*	Registers a device with the input core
`input_unregister_device()`	*drivers/input/input.c*	Removes a device from the input core
`input_report_rel()`	*include/linux/input.h*	Generates a relative movement in a specified direction
`input_report_abs()`	*include/linux/input.h*	Generates an absolute movement in a specified direction
`input_report_key()`	*include/linux/input.h*	Generates a key or a button press

Kernel Interface	Location	Description
`input_sync()`	*include/linux/input.h*	Indicates that the input subsystem can collect previously generated events into an evdev packet and send it to user space via */dev/input/inputX*
`input_register_handler()`	*drivers/input/input.c*	Registers a custom event driver
`sysfs_create_group()`	*fs/sysfs/group.c*	Creates a sysfs node group with specified attributes
`sysfs_remove_group()`	*fs/sysfs/group.c*	Removes a sysfs group created using `sysfs_create_group()`
`tty_insert_flip_char()`	*include/linux/tty_flip.h*	Sends a character to the line discipline layer
`platform_device_register_simple()`	*drivers/base/platform.c*	Creates a simple platform device
`platform_device_unregister()`	*drivers/base/platform.c*	Unregisters a platform device

Chapter 8. The Inter-Integrated Circuit Protocol

In This Chapter

- What's I²C/SMBus? 234
- I²C Core 235
- Bus Transactions 237
- Device Example: EEPROM 238
- Device Example: Real Time Clock 247
- I2C-dev 251
- Hardware Monitoring Using LM-Sensors 251
- The Serial Peripheral Interface Bus 251
- The 1-Wire Bus 254
- Debugging 254
- Looking at the Sources 255

The Inter-Integrated Circuit, or I^{2}C (pronounced *I squared C*) bus and its subset, the *System Management Bus* (SMBus), are synchronous serial interfaces that are ubiquitous on desktops and embedded devices. Let's find out how the kernel supports I^{2}C/SMBus host adapters and client devices by implementing example drivers to access an I^{2}C EEPROM and an I^{2}C RTC. And before wrapping up this chapter, let's also peek at two other serial interfaces supported by the kernel: the *Serial Peripheral Interface* or SPI (often pronounced *spy*) bus and the 1-wire bus.

All these serial interfaces (I^{2}C, SMBus, SPI, and 1-wire) share two common characteristics:

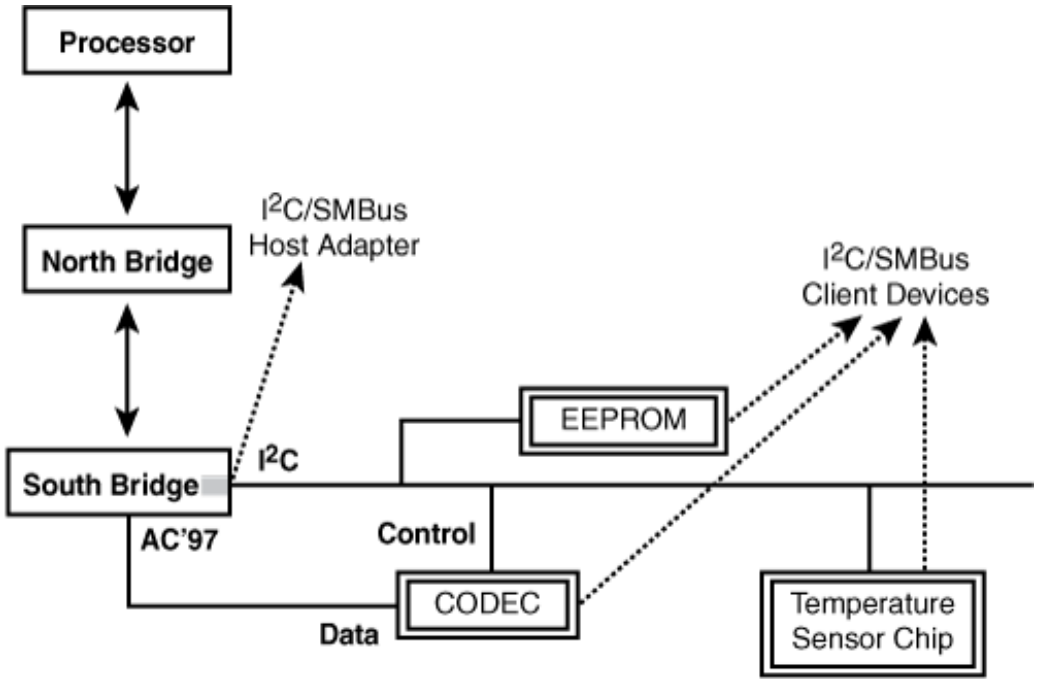
- The amount of data exchanged is small.
- The required data transfer rate is low.

What's I^{2}C/SMBus?

I^{2}C is a serial bus that is widely used in desktops and laptops to interface the processor with devices such as EEPROMs, audio codecs, and specialized chips that monitor parameters such as temperature and power-supply voltage. In addition, I^{2}C is widely used in embedded devices to communicate with RTCs, smart battery circuits, multiplexers, port expanders, optical transceivers, and other similar devices. Because I^{2}C is supported by a large number of microcontrollers, there are loads of cheap I^{2}C devices available in the market today.

I^{2}C and SMBus are master-slave protocols where communication takes place between a *host adapter* (or *host controller*) and *client devices* (or *slaves*). The host adapter is usually part of the South Bridge chipset on desktops and part of the microcontroller on embedded devices. Figure 8.1 shows an example I^{2}C bus on PC-compatible hardware.

Figure 8.1. I^{2}C/SMBus on PC-compatible hardware.



I^2C and its subset SMBus are 2-wire interfaces originally developed by Philips and Intel, respectively. The two wires are *clock* and bidirectional *data*, and the corresponding lines are called *Serial CLock* (SCL) and *Serial DAta* (SDA). Because the I^2C bus needs only a pair of wires, it consumes less space on the circuit board. However, the supported bandwidths are also low. I^2C allows up to 100Kbps in the standard mode and 400Kbps in a fast mode. (SMBus supports only up to 100Kbps, however.) The bus is thus suitable only for slow peripherals. Even though I^2C supports bidirectional exchange, the communication is half duplex because there is only a single data wire.

I^2C and SMBus devices own 7-bit addresses. The protocol also supports 10-bit addresses, but many devices respond only to 7-bit addressing, which yields a maximum of 127 devices on the bus. Due to the master-slave nature of the protocol, device addresses are also known as *slave addresses*.

Chapter 8. The Inter-Integrated Circuit Protocol

In This Chapter

- What's I²C/SMBus? 234
- I²C Core 235
- Bus Transactions 237
- Device Example: EEPROM 238
- Device Example: Real Time Clock 247
- I2C-dev 251
- Hardware Monitoring Using LM-Sensors 251
- The Serial Peripheral Interface Bus 251
- The 1-Wire Bus 254
- Debugging 254
- Looking at the Sources 255

The Inter-Integrated Circuit, or I^{2}C (pronounced *I squared C*) bus and its subset, the *System Management Bus* (SMBus), are synchronous serial interfaces that are ubiquitous on desktops and embedded devices. Let's find out how the kernel supports I^{2}C/SMBus host adapters and client devices by implementing example drivers to access an I^{2}C EEPROM and an I^{2}C RTC. And before wrapping up this chapter, let's also peek at two other serial interfaces supported by the kernel: the *Serial Peripheral Interface* or SPI (often pronounced *spy*) bus and the 1-wire bus.

All these serial interfaces (I^{2}C, SMBus, SPI, and 1-wire) share two common characteristics:

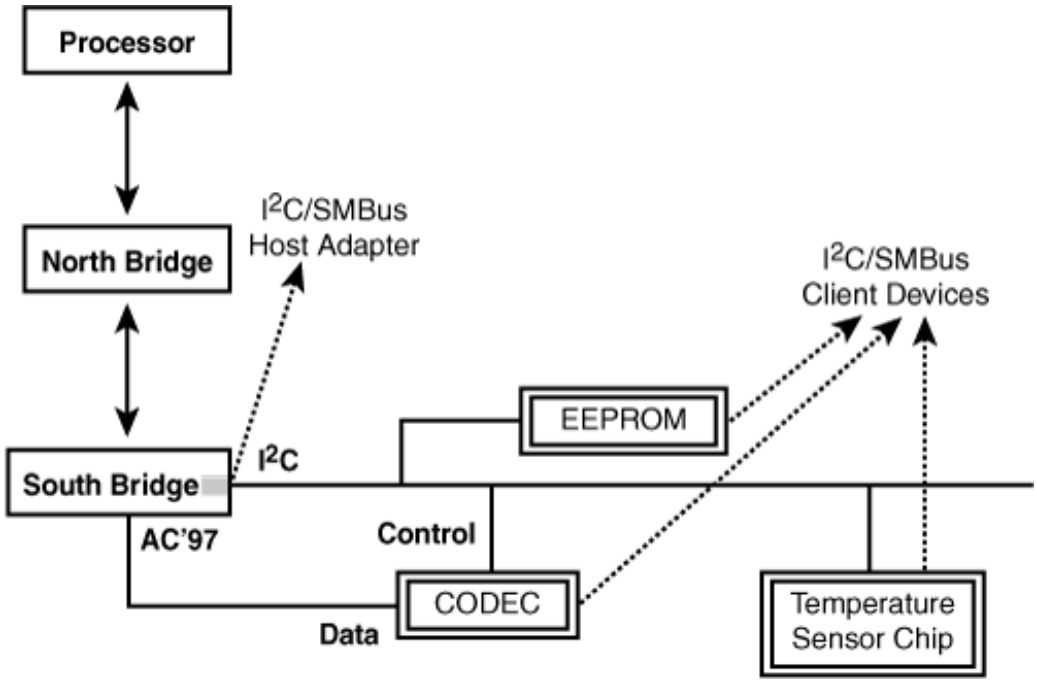
- The amount of data exchanged is small.
- The required data transfer rate is low.

What's I^{2}C/SMBus?

I^{2}C is a serial bus that is widely used in desktops and laptops to interface the processor with devices such as EEPROMs, audio codecs, and specialized chips that monitor parameters such as temperature and power-supply voltage. In addition, I^{2}C is widely used in embedded devices to communicate with RTCs, smart battery circuits, multiplexers, port expanders, optical transceivers, and other similar devices. Because I^{2}C is supported by a large number of microcontrollers, there are loads of cheap I^{2}C devices available in the market today.

I^{2}C and SMBus are master-slave protocols where communication takes place between a *host adapter* (or *host controller*) and *client devices* (or *slaves*). The host adapter is usually part of the South Bridge chipset on desktops and part of the microcontroller on embedded devices. Figure 8.1 shows an example I^{2}C bus on PC-compatible hardware.

Figure 8.1. I^{2}C/SMBus on PC-compatible hardware.



I^2C and its subset SMBus are 2-wire interfaces originally developed by Philips and Intel, respectively. The two wires are *clock* and bidirectional *data*, and the corresponding lines are called *Serial CLock* (SCL) and *Serial DAta* (SDA). Because the I^2C bus needs only a pair of wires, it consumes less space on the circuit board. However, the supported bandwidths are also low. I^2C allows up to 100Kbps in the standard mode and 400Kbps in a fast mode. (SMBus supports only up to 100Kbps, however.) The bus is thus suitable only for slow peripherals. Even though I^2C supports bidirectional exchange, the communication is half duplex because there is only a single data wire.

I^2C and SMBus devices own 7-bit addresses. The protocol also supports 10-bit addresses, but many devices respond only to 7-bit addressing, which yields a maximum of 127 devices on the bus. Due to the master-slave nature of the protocol, device addresses are also known as *slave addresses*.

I²C Core

The I²C core is a code base consisting of routines and data structures available to host adapter drivers and client drivers. Common code in the core makes the driver developer's job easier. The core also provides a level of indirection that renders client drivers independent of the host adapter, allowing them to work unchanged even if the client device is used on a board that has a different I²C host adapter. This philosophy of a core layer and its attendant benefits is also relevant for many other device driver classes in the kernel, such as PCMCIA, PCI, and USB.

In addition to the core, the kernel I²C infrastructure consists of the following:

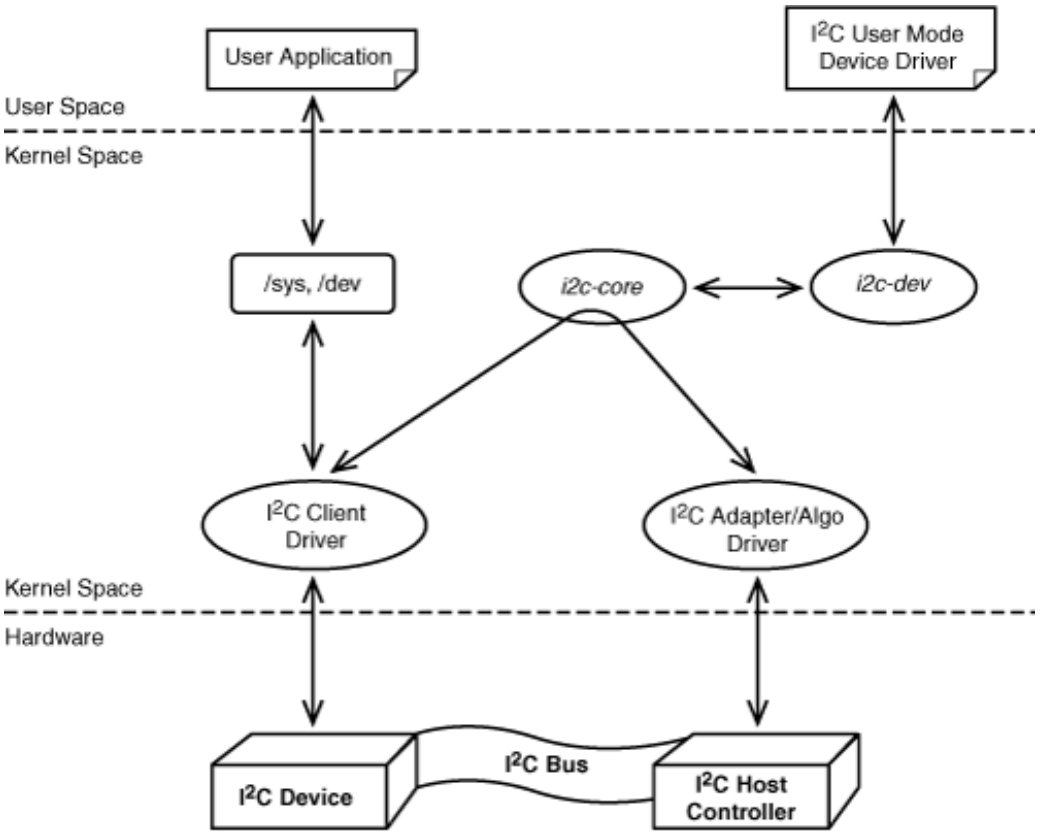
- Device drivers for I²C host adapters. They fall in the realm of bus drivers and usually consist of an *adapter* driver and an *algorithm* driver. The former uses the latter to talk to the I²C bus.
- Device drivers for I²C client devices.
- i2c-dev, which allows the implementation of user mode I²C client drivers.

You are more likely to implement client drivers than adapter or algorithm drivers because there are a lot more I²C devices than there are I²C host adapters. So, we will confine ourselves to client drivers in this chapter.

Figure 8.2 illustrates the Linux I²C subsystem. It shows I²C kernel modules talking to a host adapter and a client device on an I²C bus.

Figure 8.2. The Linux I²C subsystem.

[View full size image]



Because SMBus is a subset of I²C, using only SMBus commands to talk to your device yields a driver that works with both SMBus and I²C adapters. Table 8.1 lists the SMBus-compatible data transfer routines provided by the I²C core.

Table 8.1. SMBus-Compatible Data Access Functions Provided by the I²C Core

Function	Purpose
`i2c_smbus_read_byte()`	Reads a single byte from the device without specifying a location offset. Uses the same offset as the previously issued command.
`i2c_smbus_write_byte()`	Sends a single byte to the device at the same memory offset as the previously issued command.
`i2c_smbus_write_quick()`	Sends a single bit to the device (in place of the Rd/Wr bit shown in Listing 8.1).
`i2c_smbus_read_byte_data()`	Reads a single byte from the device at a specified offset.
`i2c_smbus_write_byte_data()`	Sends a single byte to the device at a specified offset.
`i2c_smbus_read_word_data()`	Reads 2 bytes from the specified offset.
`i2c_smbus_write_word_data()`	Sends 2 bytes to the specified offset.

Function	Purpose
i2c_smbus_read_block_data()	Reads a block of data from the specified offset.
i2c_smbus_write_block_data()	Sends a block of data (<= 32 bytes) to the specified offset.

Bus Transactions

Before implementing an example driver, let's get a better understanding of the I^{2}C protocol by peering at the wires through a magnifying glass. Listing 8.1 shows a code snippet that talks to an I^{2}C EEPROM and the corresponding transactions that occur on the bus. The transactions were captured by connecting an I^{2}C bus analyzer while running the code snippet. The code uses user mode I^{2}C functions. (We talk more about user mode I^{2}C programming in Chapter 19, "Drivers in User Space.")

Listing 8.1. Transactions on the I^{2}C Bus

Code View:

```
/* ... */
/*
 * Connect to the EEPROM. 0x50 is the device address.
 * smbus_fp is a file pointer into the SMBus device.
 */
ioctl(smbus_fp, 0x50, slave);

/* Write a byte (0xAB) at memory offset 0 on the EEPROM */
i2c_smbus_write_byte_data(smbus_fp, 0, 0xAB);

/*
 * This is the corresponding transaction observed
 * on the bus after the write:
 * S 0x50 Wr [A] 0 [A] 0xAB [A] P
 *
 * S is the start bit, 0x50 is the 7-bit slave address (0101000b),
 * Wr is the write command (0b), A is the Accept bit (or
 * acknowledgment) received by the host from the slave, 0 is the
 * address offset on the slave device where the byte is to be
 * written, 0xAB is the data to be written, and P is the stop bit.
 * The data enclosed within [] is sent from the slave to the
 * host, while the rest of the bits are sent by the host to the
 * slave.
 */
/* Read a byte from offset 0 on the EEPROM */
res = i2c_smbus_read_byte_data(smbus_fp, 0);

/*
 * This is the corresponding transaction observed
 * on the bus after the read:
 * S 0x50 Wr [A] 0 [A] S 0x50 Rd [A] [0xAB] NA P
 *
 * The explanation of the bits is the same as before, except that
 * Rd stands for the Read command (1b), 0xAB is the data received
 * from the slave, and NA is the Reverse Accept bit (or the
 * acknowledgment sent by the host to the slave).
 */
```

Device Example: EEPROM

Our first example client device is an EEPROM sitting on an I^{2}C bus, as shown in Figure 8.1. Almost all laptops and desktops have such an EEPROM for storing BIOS configuration information. The example EEPROM has two memory banks. The driver exports */dev* interfaces corresponding to each bank: */dev/eep/0* and */dev/eep/1.* Applications operate on these nodes to exchange data with the EEPROM.

Each I^{2}C/SMBus client device is assigned a slave address that functions as the device identifier. The EEPROM in the example answers to two slave addresses, SLAVE_ADDR1 and SLAVE_ADDR2, one per bank.

The example driver uses I^{2}C commands that are compatible with SMBus, so it works with both I^{2}C and SMBus EEPROMs.

Initializing

As is the case with all driver classes, I^{2}C client drivers also own an `init()` entry point. Initialization entails allocating data structures, registering the driver with the I^{2}C core, and hooking up with sysfs and the Linux device model. This is done in Listing 8.2.

Listing 8.2. Initializing the EEPROM Driver

Code View:

```
/* Driver entry points */
static struct file_operations eep_fops = {
  .owner   = THIS_MODULE,
  .llseek  = eep_llseek,
  .read    = eep_read,
  .ioctl   = eep_ioctl,
  .open    = eep_open,
  .release = eep_release,
  .write   = eep_write,
};

static dev_t dev_number;          /* Allotted Device Number */
static struct class *eep_class;   /* Device class */

/* Per-device client data structure for each
 * memory bank supported by the driver
 */

struct eep_bank {
  struct i2c_client *client;      /* I²C client for this bank */
  unsigned int addr;              /* Slave address of this bank */
  unsigned short current_pointer; /* File pointer */
  int bank_number;                /* Actual memory bank number */
  /* ... */                       /* Spinlocks, data cache for
                                     slow devices,.. */
};

#define NUM_BANKS 2               /* Two supported banks */
#define BANK_SIZE 2048            /* Size of each bank */

struct ee_bank *ee_bank_list;     /* List of private data
                                     structures, one per bank */
```

```
/*
 * Device Initialization
 */
int __init
eep_init(void)
{

  int err, i;

  /* Allocate the per-device data structure, ee_bank */
  ee_bank_list = kmalloc(sizeof(struct ee_bank)*NUM_BANKS,
                         GFP_KERNEL);
  memset(ee_bank_list, 0, sizeof(struct ee_bank)*NUM_BANKS);
  /* Register and create the /dev interfaces to access the EEPROM
     banks. Refer back to Chapter 5, "Character Drivers" for
     more details */
  if (alloc_chrdev_region(&dev_number, 0,
                          NUM_BANKS, "eep") < 0) {
    printk(KERN_DEBUG "Can't register device\n");
    return -1;
  }

  eep_class = class_create(THIS_MODULE, DEVICE_NAME);
  for (i=0; i < NUM_BANKS;i++) {

    /* Connect the file operations with cdev */
    cdev_init(&ee_bank[i].cdev, &ee_fops);

    /* Connect the major/minor number to the cdev */
    if (cdev_add(&ee_bank[i].cdev, (dev_number + i), 1)) {
      printk("Bad kmalloc\n");
      return 1;
    }
    class_device_create(eep_class, NULL, (dev_number + i),
                          NULL, "eeprom%d", i);
  }

  /* Inform the I²C core about our existence. See the section
     "Probing the Device" for the definition of eep_driver */
  err = i2c_add_driver(&eep_driver);

  if (err) {
    printk("Registering I2C driver failed, errno is %d\n", err);
    return err;
  }

  printk("EEPROM Driver Initialized.\n");
  return 0;
}
```

Listing 8.2 initiates creation of the device nodes, but to complete their production, add the following to an appropriate rule file under */etc/udev/rules.d/*:

```
KERNEL:"eeprom[0-1]*", NAME="eep/%n"
```

This creates */dev/eep/0* and */dev/eep/1* in response to reception of the corresponding uevents from the kernel. A user mode program that needs to read from the n^{th} memory bank can then operate on */dev/eep/n*.

Listing 8.3 implements the `open()` method for the EEPROM driver. The kernel calls `eep_open()` when an application opens */dev/eep/X*. `eep_open()` stores the per-device data structure in a private area so that it's directly accessible from the rest of the driver methods.

Listing 8.3. Opening the EEPROM Driver

```
 int
eep_open(struct inode *inode, struct file *file)
{

  /* The EEPROM bank to be opened */
  n = MINOR(file->f_dentry->d_inode->i_rdev);

  file->private_data = (struct ee_bank *)ee_bank_list[n];

  /* Initialize the fields in ee_bank_list[n] such as
     size, slave address, and the current file pointer */
  /* ... */
}
```

Probing the Device

The I^2C client driver, in partnership with the host controller driver and the I^2C core, attaches itself to a slave device as follows:

1. During initialization, it registers a `probe()` method, which the I^2C core invokes when an associated host controller is detected. In Listing 8.2, `eep_init()` registered `eep_probe()` by invoking `i2c_add_driver()`:

   ```
   static struct i2c_driver eep_driver =
   {
     .driver = {
       .name         =  "EEP",           /* Name */
     },
     .id             = I2C_DRIVERID_EEP, /* ID */
     .attach_adapter = eep_probe,        /* Probe Method */
     .detach_client  = eep_detach,       /* Detach Method */
   };

   i2c_add_driver(&eep_driver);  `
   ```

   The driver identifier, `I2C_DRIVERID_EEP`, should be unique for the device and should be defined in *include/linux/i2c-id.h*.

2. When the core calls the driver's `probe()` method signifying the presence of a host adapter, it, in turn, invokes `i2c_probe()` with arguments specifying the addresses of the slave devices that the driver is responsible for and an associated `attach()` routine.

Listing 8.4 implements `eep_probe()`, the `probe()` method of the EEPROM driver. `normal_i2c` specifies the EEPROM bank addresses and is populated as part of the `i2c_client_address_data` structure. Additional fields in this structure can be used to request finer addressing control. You can ask the I^2C core to ignore a range of addresses using the `ignore` field. Or you may use the `probe` field to specify (*adapter, slave address*) pairs if you want to bind a slave address to a particular host adapter. This will be useful, for example, if your processor supports two I^2C host adapters, and you have an EEPROM on bus 1 and a temperature sensor on bus 2, both answering to the same slave address.

3. The host controller walks the bus looking for the slave devices specified in Step 2. To do this, it generates a bus transaction such as `S SLAVE_ADDR Wr`, where `S` is the start bit, `SLAVE_ADDR` is the associated 7-bit slave address as specified in the device's datasheet, and `Wr` is the write command, as described in the section "Bus Transactions." If a working slave device exists on the bus, it'll respond by sending an acknowledgment bit (`[A]`).

4. If the host adapter detects a slave in Step 3, the I^2C core invokes the `attach()` routine supplied via the third argument to `i2c_probe()` in Step 2. For the EEPROM driver, this routine is `eep_attach()`, which registers a per-device client data structure, as shown in Listing 8.5. If your device expects an initial programming sequence (for example, registers on an I^2C Digital Visual Interface transmitter chip have to be initialized before the chip can start functioning), perform those operations in this routine.

Listing 8.4. Probing the Presence of EEPROM Banks

```
#include <linux/i2c.h>

/* The EEPROM has two memory banks having addresses SLAVE_ADDR1
 * and SLAVE_ADDR2, respectively
 */
static unsigned short normal_i2c[] = {
  SLAVE_ADDR1, SLAVE_ADDR2, I2C_CLIENT_END
};

static struct i2c_client_address_data addr_data = {
  .normal_i2c = normal_i2c,
  .probe      = ignore,
  .ignore     = ignore,
  .forces     = ignore,
};

static int
eep_probe(struct i2c_adapter *adapter)
{
  /* The callback function eep_attach(), is shown
   * in Listing 8.5
   */
   return i2c_probe(adapter, &addr_data, eep_attach);
}
```

Listing 8.5. Attaching a Client

```
int
eep_attach(struct i2c_adapter *adapter, int address, int kind)
{
  static struct i2c_client *eep_client;

  eep_client = kmalloc(sizeof(*eep_client), GFP_KERNEL);

  eep_client->driver  = &eep_driver; /* Registered in Listing 8.2 */
  eep_client->addr    = address;     /* Detected Address */
  eep_client->adapter = adapter;     /* Host Adapter */
  eep_client->flags   = 0;
  strlcpy(eep_client->name, "eep", I2C_NAME_SIZE);

  /* Populate fields in the associated per-device data structure */
  /* ... */

  /* Attach */
  i2c_attach_client(new_client);
}
```

Checking Adapter Capabilities

Each host adapter might be limited by a set of constraints. An adapter might not support all the commands that Table 8.1 contains. For example, it might allow the SMBus `read_word` command but not the `read_block` command. A client driver has to check whether a command is supported by the adapter before using it.

The I^2C core provides two functions to do this:

1. `i2c_check_functionality()` checks whether a particular function is supported.

2. `i2c_get_functionality()` returns a mask containing all supported functions.

See *include/linux/i2c.h* for the list of possible functionalities.

Accessing the Device

To read data from the EEPROM, first glean information about its invocation thread from the private data field associated with the device node. Next, use SMBus-compatible data access routines provided by the I^2C core (Table 8.1 shows the available functions) to read the data. Finally, send the data to user space and increment the internal file pointer so that the next `read()`/`write()` operation starts from where the last one ended. These steps are performed by Listing 8.6. The listing omits sanity and error checks for convenience.

Listing 8.6. Reading from the EEPROM

Code View:

```
ssize_t
eep_read(struct file *file, char *buf,
         size_t count, loff_t *ppos)
{
  int i, transferred, ret, my_buf[BANK_SIZE];

  /* Get the private client data structure for this bank */
  struct ee_bank *my_bank =
                  (struct ee_bank *)file->private_data;

  /* Check whether the smbus_read_word() functionality is
     supported */
  if (i2c_check_functionality(my_bank->client,
                              I2C_FUNC_SMBUS_READ_WORD_DATA)) {

    /* Read the data */
    while (transferred < count) {
      ret = i2c_smbus_read_word_data(my_bank->client,
                                     my_bank->current_pointer+i);
      my_buf[i++] = (u8)(ret & 0xFF);
      my_buf[i++] = (u8)(ret >> 8);
      transferred += 2;
    }

    /* Copy data to user space and increment the internal
       file pointer. Sanity checks are omitted for simplicity */
    copy_to_user(buffer, (void *)my_buf, transferred);
    my_bank->current_pointer += transferred;
  }

  return transferred;
}
```

Writing to the device is done similarly, except that an `i2c_smbus_write_XXX()` function is used instead.

> Some EEPROM chips have a Radio Frequency Identification (RFID) transmitter to wirelessly transmit stored information. This is used to automate supply-chain processes such as inventory monitoring and asset tracking. Such EEPROMs usually implement safeguards via an access protection bank that controls access permissions to the data banks. In such cases, the driver has to wiggle corresponding bits in the access protection bank before it can operate on the data banks.

To access the EEPROM banks from user space, develop applications that operate on */dev/eep/n.* To dump the contents of the EEPROM banks, use od:

```
bash> od -a /dev/eep/0
0000000   S   E   R   # dc4  ff soh   R   P nul nul nul nul nul nul nul
0000020   @   1   3   R   1   1   5   3   Z   J   1   V   1   L   4   6
0000040   5   1   0   H  sp   1   S   2   8   8   8   7   J   U   9   9
```

```
0000060   H   0   0   6   6 nul nul nul  bs   3   8   L   5   0   0   3
0000100   Z   J   1   N   U   B   4   6   8   6   V   7 nul nul nul nul
0000120 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
*
0000400
```

As an exercise, take a stab at modifying the EEPROM driver to create */sys* interfaces to the EEPROM banks rather than the */dev* interfaces. You may reuse code from Listing 5.7, "Using Sysfs to Control the Parallel LED Board," in Chapter 5 to help you in this endeavor.

More Methods

To obtain a fully functional driver, you need to add a few remaining entry points. These are hardly different from those of normal character drivers discussed in Chapter 5, so the code listings are not shown:

- To support the `lseek()` system call that assigns a new value to the internal file pointer, implement the `llseek()` driver method. The internal file pointer stores state information about EEPROM access.

- To verify data integrity, the EEPROM driver can support an `ioctl()` method to adjust and verify checksums of stored data.

- The `poll()` and `fsync()` methods are not relevant for the EEPROM.

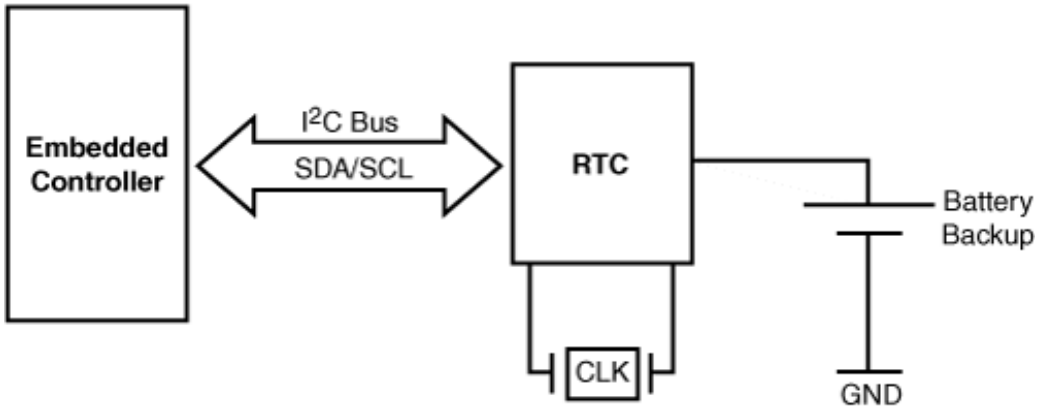
- If you choose to compile the driver as a module, you have to supply an `exit()` method to unregister the device and clean up client-specific data structures. Unregistering the driver from the I^2C core is a one-liner:

  ```
  i2c_del_driver(&eep_driver);
  ```

Device Example: Real Time Clock

Let's now take the example of an RTC chip connected to an embedded controller over the I^{2}C bus. The connection diagram is shown in Figure 8.3.

Figure 8.3. An I^{2}C RTC on an embedded system.



Assume that the I^{2}C slave address of the RTC is `0x60` and that its register space is organized as shown in Table 8.2.

Table 8.2. Register Layout on the I^{2}C RTC

Register Name	Description	Offset
`RTC_HOUR_REG`	Hour counter	`0x0`
`RTC_MINUTE_REG`	Minute counter	`0x1`
`RTC_SECOND_REG`	Second counter	`0x2`
`RTC_STATUS_REG`	Flags and interrupt status	`0x3`
`RTC_CONTROL_REG`	Enable/disable RTC	`0x4`

Let's base our driver for this chip on the EEPROM driver discussed previously. We will take the I^{2}C client driver architecture, slave registration, and I^{2}C core functions for granted and implement only the code that communicates with the RTC.

When the I^{2}C core detects a device having the RTC's slave address (`0x60`) on the I^{2}C bus, it invokes `myrtc_attach()`. The invocation train is similar to that for `eep_attach()` in Listing 8.5. Assume that you have to perform the following chip initializations in `myrtc_attach()`:

1. Clear the RTC status register (`RTC_STATUS_REG`).

2. Start the RTC (if it is not already running) by turning on appropriate bits in the RTC control register (`RTC_CONTROL_REG`).

To do this, let's build an `i2c_msg` and generate I^2C transactions on the bus using `i2c_transfer()`. This transfer mechanism is exclusive to I^2C and is not SMBus-compatible. To write to the two RTC registers referred to previously, you have to build two `i2c_msg` messages. The first message sets the register offset. In our case, it's 3, the offset of `RTC_STATUS_REG`. The second message carries the desired number of bytes to the specified offset. In this context, it ferries 2 bytes, one each to `RTC_STATUS_REG` and `RTC_CONTROL_REG`.

Code View:
```
#include <linux/i2c.h> /* For struct i2c_msg */
int
myrtc_attach(struct i2c_adapter *adapter, int addr, int kind)
{
  u8 buf[2];
  int offset = RTC_STATUS_REG;  /* Status register lives here */
  struct i2c_msg rtc_msg[2];

  /* Write 1 byte of offset information to the RTC */
  rtc_msg[0].addr = addr;       /* Slave address. In our case,
                                   this is 0x60 */
  rtc_msg[0].flags = I2C_M_WR;  /* Write Command */
  rtc_msg[0].buf = &offset;     /* Register offset for
                                   the next transaction */
  rtc_msg[0].len = 1;           /* Offset is 1 byte long */

  /* Write 2 bytes of data (the contents of the status and
     control registers) at the offset programmed by the previous
     i2c_msg */
  rtc_msg[1].addr = addr;       /* Slave address */
  rtc_msg[1].flags = I2C_M_WR;  /* Write command */
  rtc_msg[1].buf = &buf[0];     /* Data to be written to control
                                   and status registers */
  rtc_msg[1].len = 2;           /* Two register values */
  buf[0] = 0;                   /* Zero out the status register */
  buf[1] |= ENABLE_RTC;         /* Turn on control register bits
                                   that start the RTC */

  /* Generate bus transactions corresponding to the two messages */
  i2c_transfer(adapter, rtc_msg, 2);

  /* ... */
  printk("My RTC Initialized\n");
}
```

Now that the RTC is initialized and ticking, you can glean the current time by reading the contents of `RTC_HOUR_REG`, `RTC_MINUTE_REG`, and `RTC_SECOND_REG`. This is done as follows:

Code View:
```
#include <linux/rtc.h> /* For struct rtc_time */
int
```

```
myrtc_gettime(struct i2c_client *client, struct rtc_time *r_t)
{
  u8 buf[3];      /* Space to carry hour/minute/second */
  int offset = 0; /* Time-keeping registers start at offset 0 */
  struct i2c_msg rtc_msg[2];

  /* Write 1 byte of offset information to the RTC */
  rtc_msg[0].addr = addr;       /* Slave address */
  rtc_msg[0].flags = 0;         /* Write Command */
  rtc_msg[0].buf = &offset;     /* Register offset for
                                   the next transaction */
  rtc_msg[0].len = 1;           /* Offset is 1 byte long */

  /* Read current time by getting 3 bytes of data from offset 0
     (i.e., from RTC_HOUR_REG, RTC_MINUTE_REG, and RTC_SECOND_REG) */
  rtc_msg[1].addr = addr;       /* Slave address */
  rtc_msg[1].flags = I2C_M_RD;  /* Read command */
  rtc_msg[1].buf = &buf[0];     /* Data to be read from hour, minute
                                   and second registers */
  rtc_msg[1].len = 3;           /* Three registers to read */

  /* Generate bus transactions corresponding to the above
     two messages */
  i2c_transfer(adapter, rtc_msg, 2);
  /* Read the time */
  r_t->tm_hour = BCD2BIN(buf[0]); /* Hour */
  r_t->tm_min  = BCD2BIN(buf[1]); /* Minute */
  r_t->tm_sec  = BCD2BIN(buf[2]); /* Second */
  return(0);
}
```

`myrtc_gettime()` implements the bus-specific bottom layer of the RTC driver. The top layer of the RTC driver should conform to the kernel RTC API, as discussed in the section "RTC Subsystem" in Chapter 5. The advantage of this scheme is that applications can run unchanged irrespective of whether your RTC is internal to the South Bridge of a PC or externally connected to an embedded controller as in this example.

RTCs usually store time in *Binary Coded Decimal* (BCD), where each nibble represents a number between 0 and 9 (rather than between 0 and 15). The kernel provides a macro called `BCD2BIN()` to convert from BCD encoding to decimal and `BIN2BCD()` to perform the reverse operation. `myrtc_gettime()` uses the former while reading time from RTC registers.

Look at *drivers/rtc/rtc-ds1307.c* for an example RTC driver that handles the -Dallas/Maxim DS13XX series of I^2C RTC chips.

Being a 2-wire bus, the I^2C bus does not have an interrupt request line that slave devices can assert, but some I^2C host adapters have the capability to interrupt the processor to signal completion of data-transfer requests. This interrupt-driven operation is, however, transparent to I^2C client drivers and is hidden inside the service routines offered by the I^2C core. Assuming that the I^2C host controller that is part of the embedded SoC in Figure 8.3 has the capability to interrupt the processor, the invocation of `i2c_transfer()` in `myrtc_attach()` might be doing the following under the hood:

- Build a transaction corresponding to `rtc_msg[0]` and write it to the bus using the services of the host

controller device driver.

- Wait until the host controller asserts a transmit complete interrupt signaling that `rtc_msg[0]` is now on the wire.

- Inside the interrupt handler, look at the I^2C host controller status register to see whether an acknowledgment has been received from the RTC slave.

- Return error if the host controller status and control registers indicate that all's not well.

- Repeat the same for `rtc_msg[1]`.

I2C-dev

Sometimes, when you need to enable support for a large number of slow I^2C devices, it makes sense to drive them wholly from user space. The I^2C layer supports a driver called i2c-dev to achieve this. Fast forward to the section "User Mode I^2C" in Chapter 19 for an example that implements a user mode I^2C driver using i2c-dev.

Hardware Monitoring Using LM-Sensors

The *LM-Sensors* project, hosted at www.lm-sensors.org, brings hardware-monitoring capabilities to Linux. Several computer systems use sensor chips to monitor parameters such as temperature, power supply voltage, and fan speed. Periodically tracking these parameters can be critical. A blown CPU fan can manifest in the form of strange and random software problems. Imagine the consequences if the system is a medical grade device!

LM-Sensors comes to the rescue with device drivers for many sensor chips, a utility called *sensors* to generate a health report, and a script called *sensors-detect* to scan your system and help you generate appropriate configuration files.

Most chips that offer hardware monitoring interface to the CPU via I^2C/SMBus. Device drivers for such devices are normal I^2C client drivers but reside in the *drivers/hwmon/* directory, rather than *drivers/i2c/chips/*. An example is National Semiconductor's LM87 chip, which can monitor multiple voltages, temperatures, and fans. Have a look at *drivers/hwmon/lm87.c* for its driver implementation. I^2C driver IDs from 1000 to 1999 are reserved for sensor chips (look at *include/linux/i2c-id.h*).

Several sensor chips interface to the CPU via the ISA/LPC bus rather than I^2C/SMBus. Others emit analog output that reaches the CPU through an *Analog-to-Digital Converter* (ADC). Drivers for such chips share the *drivers/hwmon/* directory with I^2C sensor drivers. An example of a non-I^2C sensor driver is *drivers/hwmon/hdaps.c*, the driver for the accelerometer present in several IBM/Lenovo laptops that we discussed in Chapter 7, "Input Drivers." Another example of a non-I^2C sensor is the Winbond 83627HF Super I/O chip, which is driven by *drivers/hwmon/w83627hf.c*.

The Serial Peripheral Interface Bus

The *Serial Peripheral Interface* (SPI) bus is a serial master-slave interface similar to I^2C and comes built in on many microcontrollers. It uses four wires (compared to two on I^2C): *Serial CLocK* (SCLK), *Chip Select* (CS), *Master Out Slave In* (MOSI), and *Master In Slave Out* (MISO). MOSI is used for shifting data into the slave device, and MISO is used for shifting data out of the slave device. Because the SPI bus has dedicated wires for transmitting and receiving data, it can operate in full-duplex mode, unlike the I^2C bus. The typical speed of operation of SPI is in the low-megahertz range, unlike the mid-kilohertz range on I^2C, so the former yields higher throughput.

SPI peripherals available in the market today include *Radio Frequency* (RF) chips, smart card interfaces, EEPROMs, RTCs, touch sensors, and ADCs.

The kernel provides a core API for exchanging messages over the SPI bus. A typical SPI client driver does the following:

1. Registers `probe()` and `remove()` methods with the SPI core. Optionally registers `suspend()` and `resume()` methods:

   ```
   #include <linux/spi/spi.h>

   static struct spi_driver myspi_driver = {
     .driver  = {
       .name  = "myspi",
       .bus   = &spi_bus_type,
       .owner = THIS_MODULE,
     },
     .probe   = myspidevice_probe,
     .remove  = __devexit_p(myspidevice_remove),
   }

   spi_register_driver(&myspi_driver);
   ```

   The SPI core creates an `spi_device` structure corresponding to this device and passes this as an argument when it invokes the registered driver methods.

2. Exchanges messages with the SPI device using access functions such as `spi_sync()` and `spi_async()`. The former waits for the operation to complete, whereas the latter asynchronously triggers invocation of a registered callback routine when message transfer completes. These data access routines are invoked from suitable places such as the SPI interrupt handler, a sysfs method, or a timer handler. The following code snippet illustrates SPI message submission:

   ```
   #include <linux/spi/spi.h>

   struct spi_device *spi;  /* Representation of a
                               SPI device */
   struct spi_transfer xfer;         /* Contains transfer buffer
                                        details */
   struct spi_message sm;            /* Sequence of spi_transfer
                                        segments */
   u8 *command_buffer;               /* Data to be transferred */
   ```

```
int len;                          /* Length of data to be
                                     transferred */

spi_message_init(&sm);            /* Initialize spi_message */
xfer.tx_buf = command_buffer;     /* Device-specific data */
xfer.len = len;                   /* Data length */
spi_message_add_tail(&xfer, &sm); /* Add the message */
spi_sync(spi, &sm);               /* Blocking transfer request */
```

For an example SPI device, consider the ADS7846 touch-screen controller that we briefly discussed in Chapter 7. This driver does the following:

1. Registers `probe()`, `remove()`, `suspend()`, and `resume()` methods with the SPI core using `spi_register_driver()`.

2. The `probe()` method registers the driver with the input subsystem using `input_register_device()` and requests an IRQ using `request_irq()`.

3. The driver gathers touch coordinates from its interrupt handler using `spi_async()`. This function triggers invocation of a registered callback routine when the SPI message transfer completes.

4. The callback function in turn, reports touch coordinates and clicks via the input event interface, */dev/input/eventX*, using `input_report_abs()` and `input_report_key()`, as discussed in Chapter 7. Applications such as X Windows and gpm seamlessly work with the event interface and respond to touch input.

A driver that wiggles I/O pins to get them to talk a protocol is called a *bit-banging* driver. For an example SPI bit-banging driver, look at *drivers/spi/spi_butterfly.c*, which is a driver to talk to DataFlash chips that are present on *Butterfly* boards built by Atmel around their AVR processor family. For this, connect your host system's parallel port to the AVR Butterfly using a specially made dongle and use the *spi_butterfly* driver do the bit banging. Look at *Documentation/spi/butterfly* for a detailed description of this driver.

Currently there is no support for user space SPI drivers à la i2c-dev. You only have the option of writing a kernel driver to talk to your SPI device.

> In the embedded world, you may come across solutions where the processor uses a companion chip that integrates various functions. An example is the Freescale MC13783 Power Management and Audio Component (PMAC) used in tandem with the ARM9-based i.MX27 controller. The PMAC integrates an RTC, a battery charger, a touch-screen interface, an ADC module, and an audio codec. The processor and the PMAC communicate over SPI. The SPI bus does not contain an interrupt line, so the PMAC has the capability to externally interrupt the processor using a GPIO pin configured for this purpose.

The 1-Wire Bus

The *1-wire protocol* developed by Dallas/Maxim uses a 1-wire (or *w1*) bus that carries both power and signal; the return ground path is provided using some other means. It provides a simple way to interface with slow devices by reducing space, cost, and complexity. An example device that works using this protocol is the *ibutton* (www.ibutton.com), which is used for sensing temperature, carrying data, or holding unique IDs.

Another w1 chip that interfaces through a single port pin of an embedded controller is the DS2433 4kb 1-wire EEPROM from Dallas/Maxim. The driver for this chip, *drivers/w1/slaves/w1_ds2433.c*, exports access to the EEPROM via a sysfs node.

The main data structures associated with a w1 device driver are `w1_family` and `w1_family_ops`, both defined in *w1_family.h*.

Debugging

To collect I^{2}C-specific debugging messages, turn on a relevant combination of *I2C Core debugging messages*, *I2C Algorithm debugging messages*, *I2C Bus debugging messages*, and *I2C Chip debugging messages* under *Device Drivers* ⟶ *I2C Support* in the kernel configuration menu. Similarly, for SPI debugging, turn on *Debug Support for SPI drivers* under *Device Drivers* ⟶ *SPI Support*

To understand the flow of I^{2}C packets on the bus, connect an I^{2}C bus analyzer to your board as we did while running Listing 8.1. The lm-sensors package contains a tool called i2cdump that dumps register contents of devices on the I^{2}C bus.

There is a mailing list dedicated to Linux I^{2}C at http://lists.lm-sensors.org/mailman/listinfo/i2c.

Looking at the Sources

In the 2.4 kernel source tree, a single directory (*drivers/i2c/*) contained all the I^{2}C/SMBus-related sources. The I^{2}C code in 2.6 kernels is organized hierarchically: The *drivers/i2c/busses/* directory contains adapter drivers, the *drivers/i2c/algos/* directory has algorithm drivers, and the *drivers/i2c/chips/* directory contains client drivers. You can find client drivers in other regions of the source tree, too. The *drivers/sound/* directory, for example, includes drivers for audio chipsets that use an I^{2}C control interface. Take a look inside the *Documentation/i2c/* directory for tips and more examples.

Kernel SPI service functions live in *drivers/spi/spi.c.* The SPI driver for the ADS7846 touch controller is implemented in *drivers/input/touchscreen/ads7846.c.* The MTD subsystem discussed in Chapter 17, "Memory Technology Devices," implements drivers for SPI flash chips. An example is *drivers/mtd/devices/mtd_dataflash.c*, the driver to access Atmel DataFlash SPI chips.

The *drivers/w1/* directory contains kernel support for the w1 protocol. Drivers for the host controller side of the w1 interface live in *drivers/w1/masters/*, and drivers for w1 slaves reside in *drivers/w1/slaves/*.

Table 8.3 summarizes the main data structures used in this chapter and their location in the kernel tree. Table 8.4 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 8.3. Summary of Data Structures

Data Structure	Location	Description
`i2c_driver`	*include/linux/i2c.h*	Representation of an I^{2}C driver
`i2c_client_address_data`	*include/linux/i2c.h*	Slave addresses that an I^{2}C client driver is responsible for
`i2c_client`	*include/linux/i2c.h*	Identifies a chip connected to an I^{2}C bus
`i2c_msg`	*include/linux/i2c.h*	Information pertaining to a transaction that you want to generate on the I^{2}C bus
`spi_driver`	*include/linux/spi/spi.h*	Representation of an SPI driver
`spi_device`	*include/linux/spi/spi.h*	Representation of an SPI device
`spi_transfer`	*include/linux/spi/spi.h*	Details of an SPI transfer buffer
`spi_message`	*include/linux/spi/spi.h*	Sequence of `spi_transfer` segments
`w1_family`	*drivers/w1/w1_family.h*	Representation of a w1 slave driver
`w1_family_ops`	*drivers/w1/w1_family.h*	A w1 slave driver's entry points

Table 8.4. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`i2c_add_driver()`	*include/linux/i2c.h* *drivers/i2c/i2c-core.c*	Registers driver entry points with the I^2C core.
`i2c_del_driver()`	*drivers/i2c/i2c-core.c*	Removes a driver from the I^2C core.
`i2c_probe()`	*drivers/i2c/i2c-core.c*	Specifies the addresses of slave devices that the driver is responsible for and an associated `attach()` routine to be invoked if one of the specified addresses is detected by the I^2C core.
`i2c_attach_client()`	*drivers/i2c/i2c-core.c*	Adds a new client to the list of clients serviced by the associated I^2C host adapter.
`i2c_detach_client()`	*drivers/i2c/i2c-core.c*	Detaches an active client. Usually done when the client driver or the associated host adapter unregisters.
`i2c_check_functionality()`	*include/linux/i2c.h*	Verifies whether a particular function is supported by the host adapter.
`i2c_get_functionality()`	*include/linux/i2c.h*	Obtains a mask containing all functions supported by the host adapter.
`i2c_add_adapter()`	*drivers/i2c/i2c-core.c*	Registers a host adapter.
`i2c_del_adapter()`	*drivers/i2c/i2c-core.c*	Unregisters a host adapter.
SMBus-compatible I^2C data access routines	*drivers/i2c/i2c-core.c*	See Table 8.1.
`i2c_transfer()`	*drivers/i2c/i2c-core.c*	Sends an `i2c_msg` over the I^2C bus. This function is not SMBus-compatible.
`spi_register_driver()`	*drivers/spi/spi.c*	Registers driver entry points with the SPI core.
`spi_unregister_driver()`	*include/linux/spi/spi.h*	Unregisters an SPI driver.
`spi_message_init()`	*include/linux/spi/spi.h*	Initializes an SPI message.
`spi_message_add_tail()`	*include/linux/spi/spi.h*	Adds an SPI message to a transfer list.
`spi_sync()`	*drivers/spi/spi.c*	Synchronously transfers data over the SPI bus. This function blocks until completion.
`spi_async()`	*include/linux/spi/spi.h*	Asynchronously transfers data over the SPI bus using a completion callback mechanism.

Chapter 9. PCMCIA and Compact Flash

In This Chapter

- What's PCMCIA/CF? 258
- Linux-PCMCIA Subsystem 260
- Host Controller Drivers 262
- PCMCIA Core 263
- Driver Services 263
- Client Drivers 264
- Tying the Pieces Together 271
- PCMCIA Storage 272
- Serial PCMCIA 272
- Debugging 273
- Looking at the Sources 275

Today's popular technologies such as wireless and wired Ethernet, *General Packet Radio Service* (GPRS), *Global Positioning System* (GPS), miniature storage, and modems are ubiquitous in the form factor of PCMCIA (an acronym for *Personal Computer Memory Card International Association*) or CF (*Compact Flash*) cards. Most laptops and many embedded devices support PCMCIA or CF interfaces, thus instantly enabling them to take advantage of these technologies. On embedded systems, PCMCIA/CF slots offer a technology upgrade path without the need to re-spin the board. A cost-reduced version of an Internet-enabled device can, for example, use a PCMCIA dialup modem, while a higher-end flavor can have WiFi.

The Linux kernel supports PCMCIA devices on a variety of architectures. In this chapter, let's explore the support present in the kernel for PCMCIA/CF host adapters and client devices.

What's PCMCIA/CF?

PCMCIA is a 16-bit data-transfer interface specification originally used by memory cards. CF cards are smaller, but compatible with PCMCIA, and are frequently used in handheld devices such as PDAs and digital cameras. CF cards have only 50 pins but can be slipped into your laptop's 68-pin PCMCIA slot using a passive CF-to-PCMCIA adapter. PCMCIA and CF have been confined to the laptop and handheld space and have not made inroads into desktops and higher-end machines.

The PCMCIA specification has now grown to include support for higher speeds in the form of 32-bit *CardBus* cards. The term *PC Card* is used while referring to either PCMCIA or CardBus devices. CardBus is closer to the PCI bus, so the kernel has moved support for CardBus devices from the PCMCIA layer to the PCI layer. The latest technology specification from the PCMCIA industry standards group is the *ExpressCard*, which is compatible with *PCI Express*, a new bus technology based on PCI concepts. We look at CardBus and ExpressCard when we discuss PCI in the next chapter.

PC cards come in three flavors in the increasing order of thickness: Type I (3.3mm), Type II (5mm), and Type III (10.5mm).

Figure 9.1 shows PCMCIA bus connection on a laptop, and Figure 9.2 illustrates PCMCIA on an embedded device. As you might have noticed, the PCMCIA host controller bridges the PCMCIA card with the system bus. Laptops and their derivatives generally have a PCMCIA host controller chip connected to the PCI bus, while several embedded controllers have a PCMCIA host controller built in to their silicon. The controller maps card memory to host I/O and memory windows and routes interrupts generated by the card to a suitable processor interrupt line.

Figure 9.1. PCMCIA on a laptop.

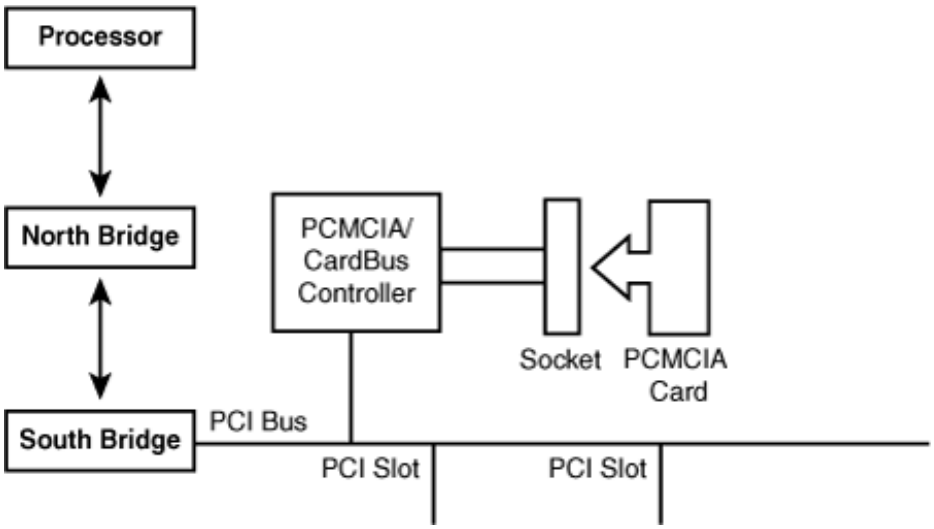
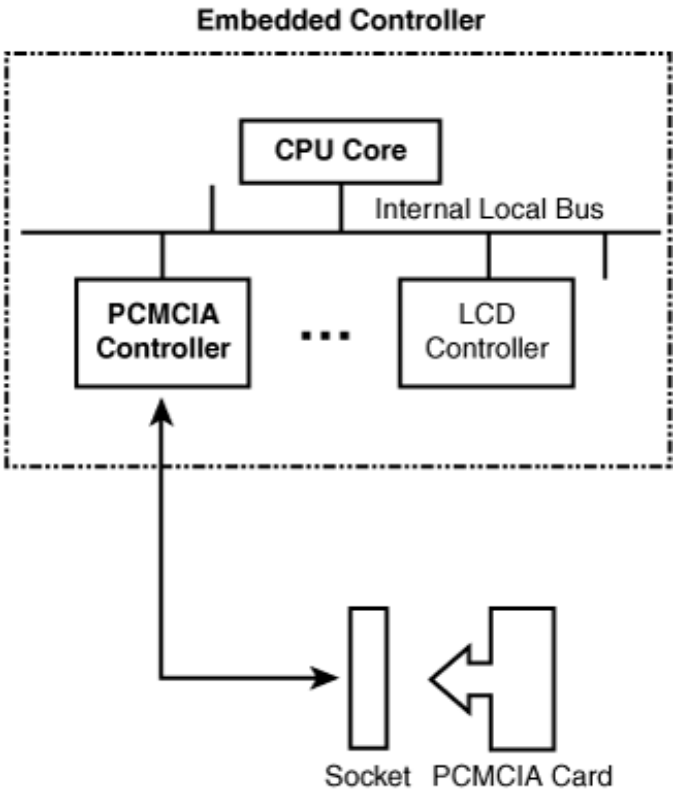


Figure 9.2. PCMCIA on an embedded system.



Chapter 9. PCMCIA and Compact Flash

In This Chapter

- What's PCMCIA/CF? 258
- Linux-PCMCIA Subsystem 260
- Host Controller Drivers 262
- PCMCIA Core 263
- Driver Services 263
- Client Drivers 264
- Tying the Pieces Together 271
- PCMCIA Storage 272
- Serial PCMCIA 272
- Debugging 273
- Looking at the Sources 275

Today's popular technologies such as wireless and wired Ethernet, *General Packet Radio Service* (GPRS), *Global Positioning System* (GPS), miniature storage, and modems are ubiquitous in the form factor of PCMCIA (an acronym for *Personal Computer Memory Card International Association*) or CF (*Compact Flash*) cards. Most laptops and many embedded devices support PCMCIA or CF interfaces, thus instantly enabling them to take advantage of these technologies. On embedded systems, PCMCIA/CF slots offer a technology upgrade path without the need to re-spin the board. A cost-reduced version of an Internet-enabled device can, for example, use a PCMCIA dialup modem, while a higher-end flavor can have WiFi.

The Linux kernel supports PCMCIA devices on a variety of architectures. In this chapter, let's explore the support present in the kernel for PCMCIA/CF host adapters and client devices.

What's PCMCIA/CF?

PCMCIA is a 16-bit data-transfer interface specification originally used by memory cards. CF cards are smaller, but compatible with PCMCIA, and are frequently used in handheld devices such as PDAs and digital cameras. CF cards have only 50 pins but can be slipped into your laptop's 68-pin PCMCIA slot using a passive CF-to-PCMCIA adapter. PCMCIA and CF have been confined to the laptop and handheld space and have not made inroads into desktops and higher-end machines.

The PCMCIA specification has now grown to include support for higher speeds in the form of 32-bit *CardBus* cards. The term *PC Card* is used while referring to either PCMCIA or CardBus devices. CardBus is closer to the PCI bus, so the kernel has moved support for CardBus devices from the PCMCIA layer to the PCI layer. The latest technology specification from the PCMCIA industry standards group is the *ExpressCard*, which is compatible with *PCI Express*, a new bus technology based on PCI concepts. We look at CardBus and ExpressCard when we discuss PCI in the next chapter.

PC cards come in three flavors in the increasing order of thickness: Type I (3.3mm), Type II (5mm), and Type III (10.5mm).

Figure 9.1 shows PCMCIA bus connection on a laptop, and Figure 9.2 illustrates PCMCIA on an embedded device. As you might have noticed, the PCMCIA host controller bridges the PCMCIA card with the system bus. Laptops and their derivatives generally have a PCMCIA host controller chip connected to the PCI bus, while several embedded controllers have a PCMCIA host controller built in to their silicon. The controller maps card memory to host I/O and memory windows and routes interrupts generated by the card to a suitable processor interrupt line.

Figure 9.1. PCMCIA on a laptop.

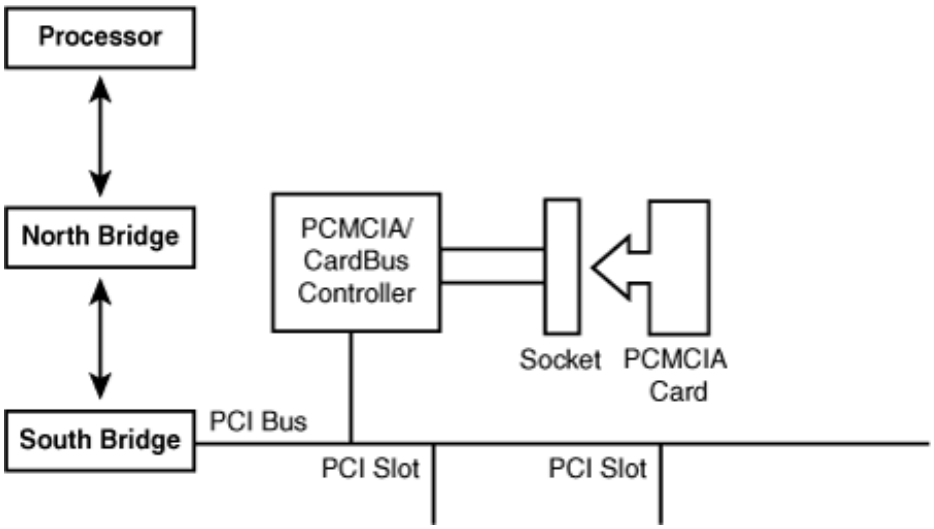
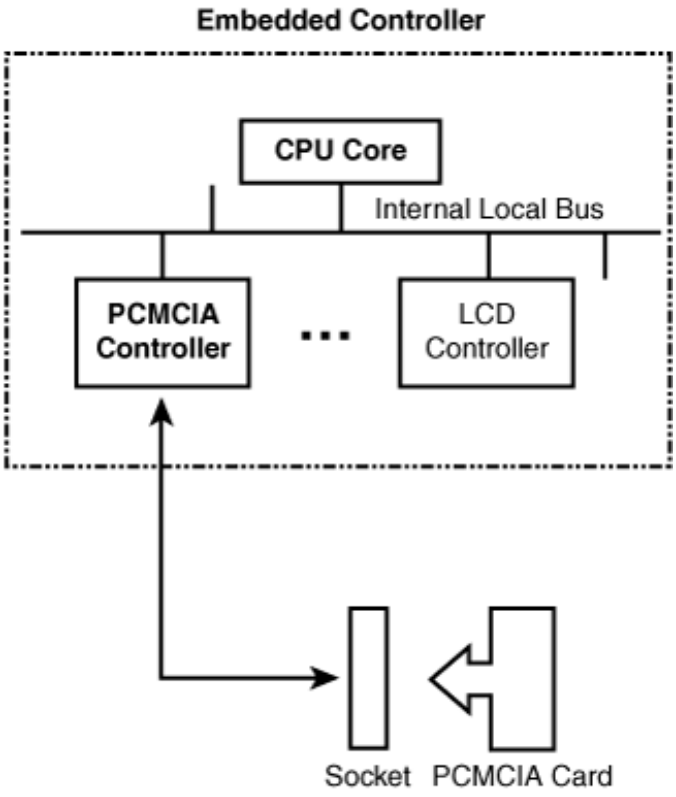


Figure 9.2. PCMCIA on an embedded system.



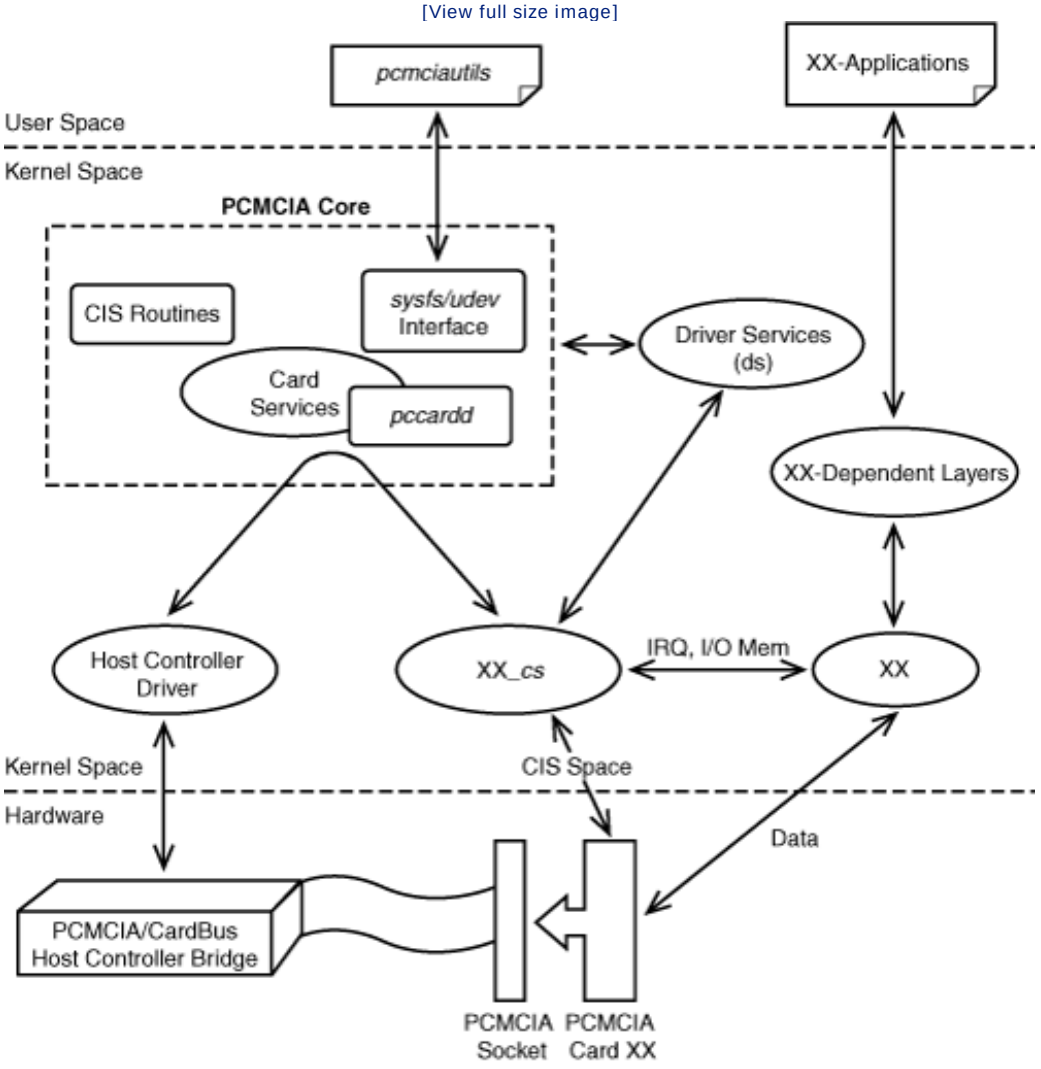
Linux-PCMCIA Subsystem

Linux-PCMCIA support is available on Intel-based laptops as well as on architectures such as ARM, MIPS, and PowerPC. The PCMCIA subsystem consists of device drivers for PCMCIA host controllers, client drivers for different cards, a daemon that aids hotplugging, user mode utilities, and a *Card Services* module that interacts with all of these.

Figure 9.3 illustrates the interaction between the modules that constitute the Linux-PCMCIA subsystem.

Figure 9.3. The Linux-PCMCIA subsystem.

[View full size image]



The Old Linux-PCMCIA Subsystem

The Linux-PCMCIA subsystem has recently undergone an overhaul. To get PCMCIA working with 2.6.13 and newer kernels, you need the *pcmciautils* package (http://kernel.org/pub/linux/utils/kernel/pcmcia/howto.html), which obsoletes the *pcmcia-cs* package (http://pcmcia-cs.sourceforge.net) used with earlier kernels. Internal kernel programming interfaces and data structures have also changed. Earlier kernels relied on a user space daemon called *cardmgr* to support hotplugging, but the new PCMCIA implementation handles hotplug using *udev*, just as other bus subsystems do. So with new setups, you don't need *cardmgr* and should make sure that it is not started. There is a migration guide at http://kernel.org/pub/linux/utils/kernel/pcmcia/cardmgr-to-pcmciautils.html.

Figure 9.3 contains the following components:

- Host controller device drivers that implement low-level routines for communicating with the PCMCIA host controller. Your handheld and laptop have different host controllers and, hence, use different host controller drivers. Each PCMCIA slot that the host controller supports is called a *socket*.

- PCMCIA client drivers (XX_cs in Figure 9.3) that respond to socket events such as card insertion and ejection. This is the driver that you are most likely to implement when you attempt to Linux-enable a PCMCIA card. The XX_cs driver usually works in tandem with a generic driver (XX in Figure 9.3) that is not PCMCIA-specific. In relation to Figure 9.3, if your device is a PCMCIA IDE disk, XX is the IDE disk driver, XX_cs is the ide_cs driver, XX-dependent layers are filesystem layers, and XX-applications are programs that access data files. XX_cs configures the generic driver (XX) with resources such as IRQs, I/O base addresses, and memory windows.

- The PCMCIA core that provides services to host controller drivers and client drivers. The core provides an infrastructure that makes driver implementations simpler and adds a level of indirection that renders client drivers independent of host controllers. Irrespective of whether you are using your Bluetooth CF card on an XScale-based handheld or an x86-based laptop, the same client drivers can be pressed into service.

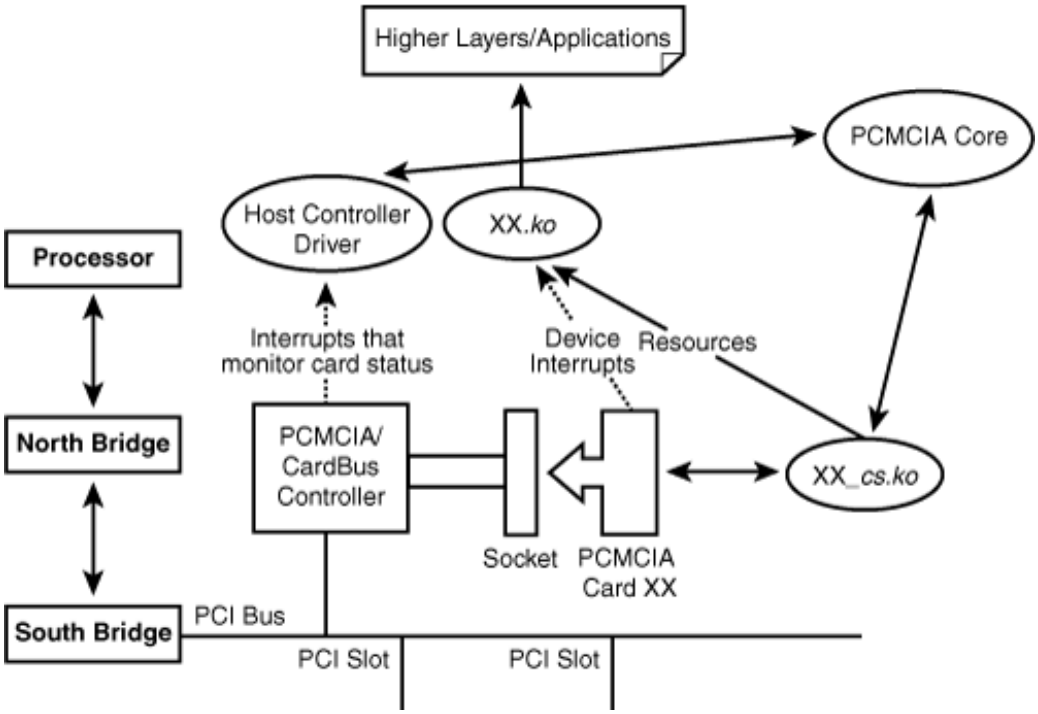
- A *driver services* module (*ds*) that offers registration interfaces and bus services to client drivers.

- The *pcmciautils* package, which contains tools such as *pccardctl* that control the state of PCMCIA sockets and select between different card-configuration schemes.

Figure 9.4 glues kernel modules on top of Figure 9.1 to illustrate how the Linux-PCMCIA subsystem interacts with hardware on a PC-compatible system.

Figure 9.4. Relating PCMCIA driver components with PC hardware.

[View full size image]



In the following sections, let's take a closer look at the components constituting the Linux-PCMCIA subsystem. To better understand the role of these components and their interaction, we will insert a PCMCIA WiFi card into a laptop and trace the code flow in the section "Tying the Pieces Together."

Host Controller Drivers

Whereas the generic card driver (XX) is responsible for handling interrupts generated by the card function (say, receive interrupts when a PCMCIA network card receives data packets), the host controller driver is responsible for handling bus-specific interrupts triggered by events such as card insertion and ejection.

Figure 9.2 shows the block diagram of an embedded device designed around an embedded controller that has built-in PCMCIA support. Even if you are using a controller supported by the kernel PCMCIA layer, you might need to tweak the host controller driver (for example, to configure GPIO lines used for detecting card insertion events or switching power to the socket) depending on your board's design. If you are porting the kernel to a StrongARM-based handheld, for example, tailor *drivers/pcmcia/sa1100_assabet.c* to suit your hardware.

This chapter does not cover the implementation of host controller device drivers.

PCMCIA Core

Card Services is the main constituent of the PCMCIA core. It offers a set of services to client drivers and host controller drivers. It contains a kernel thread called *pccardd* that polls for socket-related events. Pccardd notifies the Driver Services event handler (discussed in the next section) when the host controller reports events such as card insertion and card removal.

Another component of the PCMCIA core is a library that manipulates the *Card Information Structure* (CIS) that is part of PCMCIA cards. PCMCIA/CF cards have two memory spaces: *Attribute* memory and *Common* memory. Attribute memory contains the CIS and card configuration registers. Attribute memory of a PCMCIA IDE disk, for example, contains its CIS and registers that specify the sector count and the cylinder number. Common memory in this case contains the memory array that holds disk data. The PCMCIA core offers CIS manipulation routines such as `pccard_get_first_tuple()`, `pccard_get_next_tuple()`, and `pccard_parse_tuple()` to client drivers. Listing 9.2 uses the assistance of some of these functions.

The PCMCIA core passes CIS information to user space via sysfs and udev. Utilities such as pccardctl, part of the pcmciautils package, depend on sysfs and udev for their operation. This simplifies the earlier design approach that relied on a custom infrastructure when these facilities were absent in the kernel.

Driver Services

Driver Services provides an infrastructure that offers the following:

- A handler that catches event alerts dispatched by the pccardd kernel thread. The handler scans and validates the card's CIS space and triggers the load of an appropriate client driver.

- A layer that has the task of communicating with the kernel's bus core. To this end, Driver Services implements the `pcmcia_bus_type` and related bus operations.

- Service routines such as `pcmcia_register_driver()` that client drivers use to register themselves with the PCMCIA core. The example driver in Listing 9.1 uses some of these routines.

Client Drivers

The client device driver (XX_cs in Figure 9.3) looks at the card's CIS space and configures the card depending on the information it gathers.

Data Structures

Before proceeding to develop an example PCMCIA client driver, let's meet some related data structures:

1. A PCMCIA device is identified by the `pcmcia_device_id` structure defined in *include/linux/mod_devicetable.h*:

   ```
   struct pcmcia_device_id {
     /* ... */
     __u16       manf_id;       /* Manufacturer ID */
     __u16       card_id;       /* Card ID */
     __u8        func_id;       /* Function ID */
     /* ... */
   };
   ```

   `manf_id`, `card_id`, and `func_id` hold the card's manufacturer ID, card ID, and function ID, respectively. The PCMCIA core offers a macro called `PCMCIA_DEVICE_MANF_CARD()` that creates a `pcmcia_device_id` structure from the manufacturer and card IDs supplied to it. Another kernel macro called `MODULE_DEVICE_TABLE()` marks the supported `pcmcia_device_id`s in the module image so that the module can be loaded on demand when the card is inserted and the PCMCIA subsystem gleans matching manufacturer/card/function IDs from the card's CIS space. We explored this mechanism in the section "Module Autoload" in Chapter 4, "Laying the Groundwork." This procedure is analogous to that used by device drivers for two other popular I/O buses that support hotplugging: PCI and USB. Table 9.1 gives a heads-up on the similarities between drivers for these three bus technologies. Don't worry if that is hard to digest; we will have a detailed discussion on PCI and USB in the following chapters.

Table 9.1. Device IDs and Hotplug Methods for PCMCIA, PCI, and USB

	PCMCIA	PCI	USB
Device ID table structure	`pcmcia_device_id`	`pci_device_id`	`usb_device_id`
Macro to create a device ID	`PCMCIA_DEVICE_MANF_CARD()`	`PCI_DEVICE()`	`USB_DEVICE()`
Device representation	`struct pcmcia_device`	`struct pci_dev`	`struct usb_device`
Driver representation	`struct pcmcia_driver`	`struct pci_driver`	`struct usb_driver`
Hotplug methods	`probe()` and `remove()`	`probe()` and `remove()`	`probe()` and `disconnect()`
Hotplug event detection	pccardd kthread	PCI-family-dependent	khubd kthread

2. PCMCIA client drivers need to associate their `pcmcia_device_id` table with their `probe()` and `remove()` methods. This tie up is achieved by the `pcmcia_driver` structure:

```
struct pcmcia_driver {
  int  (*probe)(struct pcmcia_device *dev);  /* Probe
                                                method */
  void (*remove)(struct pcmcia_device *dev); /* Remove
                                                method */
  /* ... */
  struct pcmcia_device_id *id_table;         /* Device ID
                                                table */
  /* ... */
};
```

3. `struct pcmcia_device` internally represents a PCMCIA device and is defined as follows in *drivers/pcmcia/ds.h*:

```
struct pcmcia_device {
  /* ... */
  io_req_t io;          /* I/O attributes*/
  irq_req_t irq;        /* IRQ settings */
  config_req_t conf;    /* Configuration */
  /* ... */
  struct device dev;    /* Connection to device model */
  /* ... */
};
```

4. CIS manipulation routines use a `tuple_t` structure defined in *include/pcmcia/cistpl.h* to hold a CIS information unit. A `CISTPL_LONGLINK_MFC` tuple type, for example, contains information related to a multifunction card. For the full list of tuples and their descriptions, look at *include/pcmcia/cistpl.h* and http://pcmcia-cs.sourceforge.net/ftp/doc/PCMCIA-PROG.html.

```
typedef struct tuple_t {
  /* ... */
  cisdata_t TupleCode;     /* See
                              include/pcmcia/cistpl.h */
  /* ... */
  cisdata_t DesiredTuple;  /* Identity of the desired
                              tuple */
  /* ... */
  cisdata_t *TupleData;    /* Buffer space */
};
```

5. The CIS contains configuration table entries for each configuration that the card supports. `cistpl_cftable_entry_t`, defined in *include/pcmcia/cistpl.h*, holds such an entry:

```
typedef struct cistpl_cftable_entry_t {
  /* ... */
  cistpl_power_t vcc, vpp1, vpp2;   /* Voltage level */
  cistpl_io_t io;                   /* I/O attributes */
  cistpl_irq_t irq;                 /* IRQ settings */
  cistpl_mem_t mem;                 /* Memory window */
  /* ... */
};
```

6. `cisparse_t`, also defined in *include/pcmcia/cistpl.h*, holds a tuple parsed by the PCMCIA core:

```
typedef union cisparse_t {
  /* ... */
  cistpl_manfid_t manfid;                 /* Manf ID */
  /* ... */
  cistpl_cftable_entry_t cftable_entry;   /* Configuration
                                             table entry */
  /* ... */
} cisparse_t;
```

Device Example: PCMCIA Card

Let's develop a skeletal client device driver (because too many details will make it a loaded discussion) to learn the workings of the PCMCIA subsystem. The implementation is general, so you may use it as a template irrespective of whether your card implements networking, storage, or some other technology. Only the XX_cs driver is implemented; the generic XX driver is assumed to be available off the shelf.

As alluded to earlier, PCMCIA drivers contain `probe()` and `remove()` methods to support hotplugging. Listing 9.1 registers the driver's `probe()` method, `remove()` method, and `pcmcia_device_id` table with the PCMCIA core. `XX_probe()` gets invoked when the associated PCMCIA card is inserted, and `XX_remove()` is called when the card is ejected.

Listing 9.1. Registering a Client Driver

Code View:

```
#include <pcmcia/ds.h> /* Definition of struct pcmcia_device */

static struct pcmcia_driver XX_cs_driver = {
  .owner       = THIS_MODULE,
  .drv         = {
  .name = "XX_cs",             /* Name */
  },
  .probe       = XX_probe,     /* Probe */
  .remove      = XX_remove,    /* Release */
  .id_table    = XX_ids,       /* ID table */
  .suspend     = XX_suspend,   /* Power management */
  .resume      = XX_resume,    /* Power management */
};

#define XX_MANFUFACTURER_ID  0xABCD  /* Device's manf_id */
#define XX_CARD_ID           0xCDEF  /* Device's card_id */

/* Identity of supported cards */
static struct pcmcia_device_id XX_ids[] = {
  PCMCIA_DEVICE_MANF_CARD(XX_MANFUFACTURER_ID, XX_CARD_ID),
  PCMCIA_DEVICE_NULL,
};

MODULE_DEVICE_TABLE(pcmcia, XX_ids); /* For module autoload */

/* Initialization */
static int __init
init_XX_cs(void)
{
  return pcmcia_register_driver(&XX_cs_driver);
}
```

```
/* Probe Method */
static int
XX_probe(struct pcmcia_device *link)
{
  /* Populate the pcmcia_device structure allotted for this card by
     the core. First fill in general information */
  /* ... */

  /* Fill in attributes related to I/O windows and
     interrupt levels */
  XX_config(link); /* See Listing 9.2 */
}
```

Listing 9.2 shows the routine that configures the generic device driver (XX) with resource information such as I/O and memory window base addresses. After this step, data flow to and from the PCMCIA card passes through XX and is transparent to the rest of the layers. Any interrupts generated by the PCMCIA card, such as those related to data reception or transmit completion for network cards, are handled by the interrupt handler that is part of XX. Listing 9.2 is loosely based on *drivers/net/wireless/airo_cs.c*, the client driver for the Cisco Aironet 4500 and 4800 series of PCMCIA WiFi cards. The listing uses the services of the PCMCIA core to do the following:

- Obtain a suitable configuration table entry tuple from the card's CIS
- Parse the tuple
- Glean card configuration information such as I/O base addresses and power settings from the parsed tuple
- Request allocation of an interrupt line

It then configures the chipset-specific driver (XX) with the information previously obtained.

Listing 9.2. Configuring the Generic Device Driver

Code View:

```
#include <pcmcia/cistpl.h>
#include <pcmcia/ds.h>
#include <pcmcia/cs.h>
#include <pcmcia/cisreg.h>

/* This makes the XX device available to the system. XX_config()
   is based on airo_config(), defined in
   drivers/net/wireless/airo_cs.c */
static int
XX_config(struct pcmcia_device *link)
{
  tuple_t tuple;
  cisparse_t parse;
  u_char buf[64];

  /* Populate a tuple_t structure with the identity of the desired
     tuple. In this case, we're looking for a configuration table
```

```
   entry */
tuple.DesiredTuple = CISTPL_CFTABLE_ENTRY;
tuple.Attributes   = 0;
tuple.TupleData    = buf;
tuple.TupleDataMax = sizeof(buf);

/* Walk the CIS for a matching tuple and glean card configuration
   information such as I/O window base addresses */

/* Get first tuple */
CS_CHECK(GetFirstTuple, pcmcia_get_first_tuple(link, &tuple));
while (1){
  cistpl_cftable_entry_t dflt = {0};
  cistpl_cftable_entry_t *cfg = &(parse.cftable_entry);

  /* Read a configuration tuple from the card's CIS space */
  if (pcmcia_get_tuple_data(link, &tuple) != 0 ||
      pcmcia_parse_tuple(link, &tuple, &parse) != 0) {
    goto next_entry;
  }

  /* We have a matching tuple! */
  /* Configure power settings in the pcmcia_device based on
     what was found in the parsed tuple entry */
  if (cfg->vpp1.present & (1<<CISTPL_POWER_VNOM))
    link->conf.Vpp = cfg->vpp1.param[CISTPL_POWER_VNOM]/10000;

  /* ... */

  /* Configure I/O window settings in the pcmcia_device based on
     what was found in the parsed tuple entry */
  if ((cfg->io.nwin > 0) || (dflt.io.nwin > 0)) {
    cistpl_io_t *io = (cfg->io.nwin) ? &cfg->io : &dflt.io;
    /* ... */
    if (!(io->flags & CISTPL_IO_8BIT)) {
      link->io.Attributes1 = IO_DATA_PATH_WIDTH_16;
    }
    link->io.BasePort1 = io->win[0].base;
    /* ... */
  }

  /* ... */
  break;
next_entry:
  CS_CHECK(GetNextTuple, pcmcia_get_next_tuple(link, &tuple);
}

/* Allocate IRQ */
if (link->conf.Attributes & CONF_ENABLE_IRQ) {
  CS_CHECK(RequestIRQ, pcmcia_request_irq(link, &link->irq));
}
/* ... */

/* Invoke init_XX_card(), which is part of the generic
   XX driver (so, not shown in this listing), and pass
   the I/O base and IRQ information obtained above */
init_XX_card(link->irq.AssignedIRQ, link->io.BasePort1,
             1, &handle_to_dev(link));
```

```
  /* The chip-specific (form factor independent) driver is ready
     to take responsibility of this card from now on! */
}
```

Tying the Pieces Together

As you saw in Figure 9.3, the PCMCIA layer consists of various components. The data-flow path between the components can sometimes get complicated. Let's trace the code path from the time you insert a PCMCIA card until an application starts transferring data to the card. Assume that a Cisco Aironet PCMCIA card is inserted onto a laptop having an 82365-compatible PCMCIA host controller:

1. The PCMCIA host controller driver (*drivers/pcmcia/yenta_socket.c*) detects the insertion event via its interrupt service routine and makes note of it using suitable data structures.

2. The pccardd kernel thread that is part of Card Services (*drivers/pcmcia/cs.c*) sleeps on a wait queue until the host controller driver wakes it up when it detects the card insertion in Step 1.

3. Card Services dispatches an insertion event to Driver Services (*drivers/pcmcia/ds.c*). This triggers execution of the event handler registered by Driver Services during initialization.

4. Driver Services validates the card's CIS space, determines information about the inserted device such as its manufacturer ID and card ID, and registers the device with the kernel. The appropriate client device driver (*drivers/net/wireless/airo_cs.c*) is then loaded. Revisit our previous discussion on `MODULE_DEVICE_TABLE()` to see how this is accomplished.

5. The client driver (*airo_cs.c*) loaded in Step 4 initializes and registers itself using `pcmcia_register_driver()`, as shown in Listing 9.1. This registration interface internally sets the bus type of the device to `pcmcia_bus_type`. PCMCIA bus operations such as `probe()` and `remove()`, defined by Driver Services (*ds.c*), are also internally registered.

6. The kernel invokes the bus `probe()` operation registered by Driver Services in Step 5. This in turn, invokes the `probe()` method owned by the matching client driver (`airo_probe()`), also registered in Step 5. The client `probe()` routine populates settings, such as I/O windows and interrupt lines, and configures the generic chipset-specific driver (*drivers/net/wireless/airo.c*), as shown in Listing 9.2.

7. The chipset driver (*airo.c*) creates a network interface (ethX) and is responsible for normal operation from this point onward. It's this driver that handles interrupts generated by the card in response to packet reception and transmit completion. The form factor of the device (for example, whether it's a PCMCIA or a PCI card) is transparent to the chipset driver as well as to the applications that operate over ethX.

PCMCIA Storage

Today's PCMCIA/CF storage support densities in the gigabyte realm. The storage cards come in different flavors:

- Miniature IDE disk drives or microdrives. These are tiny versions of mechanical hard drives that use magnetic media. Their data transfer rates are typically higher than solid state memory devices, but IDE drives have spin-up and seek latencies before data can be transferred. The IDE Card Services driver *ide_cs*, in conjunction with legacy IDE drivers, is used to communicate with such memory cards.

- Solid-state memory cards that emulate IDE. Such cards have no moving parts and are usually based on flash memory, which is transparent to the operating system because of the IDE emulation. Because these drives are effectively IDE-based, the same IDE Card Services driver (ide_cs) can be used to talk to them.

- Memory cards that use flash memory, but without IDE emulation. The *memory_cs* Card Services driver provides block and character interfaces over such cards. The block interface is used to put a filesystem onto card memory, whereas the character interface is used to access raw data. You may also use memory_cs to read the attribute memory space of any PCMCIA card.

Serial PCMCIA

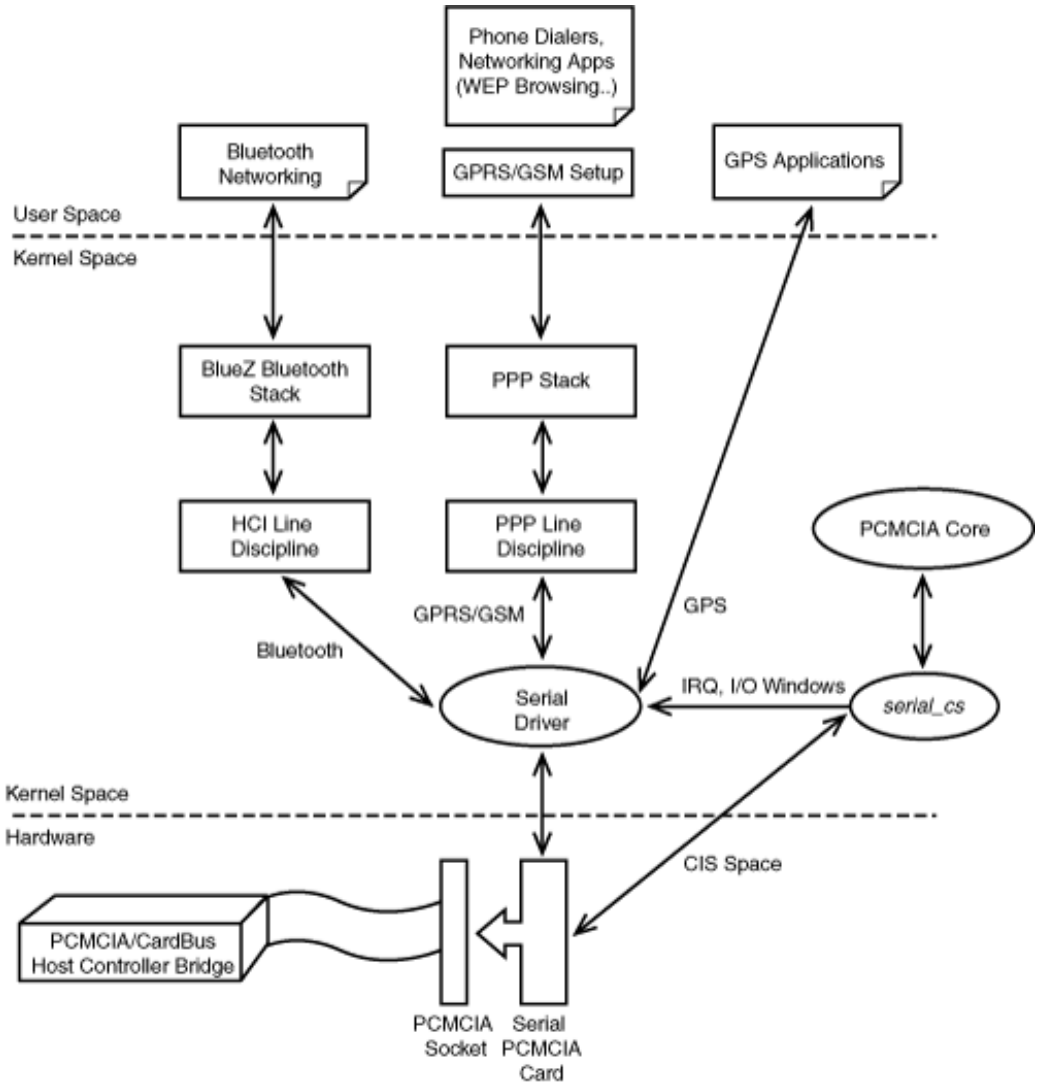
Many networking technologies such as *General Packet Radio Service* (GPRS), *Global System for Mobile Communications* (GSM), *Global Positioning System* (GPS), and Bluetooth use a serial transport mechanism to communicate with host systems. In this section, let's find out how the PCMCIA layer handles cards that feature such technologies. Note that this section is only to help you understand the bus interface part of GPRS, GSM, and Bluetooth cards having a PCMCIA/CF form factor. The technologies themselves are discussed in detail in Chapter 16, "Linux Without Wires."

The generic serial Card Services driver, *serial_cs*, allows the rest of the operating system to see the PCMCIA/CF card as a serial device. The first unused serial device, */dev/ttySX*, gets allotted to the card. serial_cs thus emulates a serial port over GPRS, GSM, and GPS cards. It also allows Bluetooth PCMCIA/CF cards that use a serial transport to transfer *Host Control Interface* (HCI) packets to Bluetooth protocol layers.

Figure 9.5 illustrates how kernel modules implementing different networking technologies interact with serial_cs to communicate with their respective cards.

Figure 9.5. Networking with PCMCIA/CF cards that use serial transport.

[View full size image]



The *Point-to-Point Protocol* (PPP) allows networking protocols such as TCP/IP to run over a serial link. In the context of Figure 9.5, PPP gets TCP/IP applications running over GPRS and GSM dialup. The PPP daemon, *pppd*, attaches over virtual serial ports emulated by serial_cs. The PPP kernel modules—*ppp_generic, ppp_async*, and *slhc*—have to be loaded for pppd to work. Invoke pppd as follows:

```
bash> pppd ttySX call connection-script
```

where *connection-script* is a file containing command sequences that pppd exchanges with the service provider to establish a link. The connection script depends on the particular card that is being used. A GPRS card would need a context string to be sent as part of the connection script, whereas a GSM card might need an exchange of passwords. An example connection script is described in the section "GPRS" in Chapter 16.

Debugging

To effectively debug PCMCIA/CF client drivers, you need to see debug messages emitted by the PCMCIA core. For this, enable `CONFIG_PCMCIA_DEBUG` (*Bus options* ⟶ *PCCARD support* ⟶ *Enable PCCARD debugging*) during kernel configuration. Verbosity levels of the debug output can be controlled either via the `pcmcia_core.pc_debug` kernel command-line argument or using the `pc_debug` module insertion parameter.

Information about PC Card client drivers is available in the process filesystem entry, */proc/bus/pccard/drivers*. Look at */sys/bus/pcmcia/devices/** for card-specific information such as manufacturer and card IDs. Take a look inside */proc/bus/pci/* to know more about your PCMCIA host controller if your system uses a PCI-to-PCMCIA bridge. */proc/interrupts* lists IRQs active on your system, including those used by the PCMCIA layer.

There is a mailing list dedicated to Linux-PCMCIA at http:///lists.infradead.org/mailman/listinfo/linux-pcmcia.

Looking at the Sources

In the Linux source tree, the *drivers/pcmcia/* directory contains the sources for Card Services, Driver Services, and host controller drivers. Look at *drivers/pcmcia/yenta_socket.c* for the host controller driver that runs on many x86-based laptops. Header files present in *include/pcmcia/* contain PCMCIA-related structure definitions.

Client drivers live alongside other drivers belonging to the associated device class. So, you will find drivers for PCMCIA networking cards inside *drivers/net/pcmcia/*. The client driver for PCMCIA memory devices that emulate IDE is *drivers/ide/legacy/ ide-cs.c*. See *drivers/serial/serial_cs.c* for the client driver used by PCMCIA modems.

Table 9.2 summarizes the main data structures used in this chapter and their location in the kernel tree. Table 9.3 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 9.2. Summary of Data Structures

Data Structure	Location	Description
`pcmcia_device_id`	*include/linux/mod_devicetable.h*	Identity of a PCMCIA card.
`pcmcia_device`	*include/pcmcia/ds.h*	Representation of a PCMCIA device.
`pcmcia_driver`	*include/pcmcia/ds.h*	Representation of a PCMCIA client driver.
`tuple_t`	*include/pcmcia/cistpl.h*	CIS manipulation routines use a `tuple_t` structure to hold information.
`cistpl_cftable_entry_t`	*include/pcmcia/cistpl.h*	Configuration table entry in the CIS space.
`cisparse_t`	*include/pcmcia/cistpl.h*	A parsed CIS tuple.

Table 9.3. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`pcmcia_register_driver()`	*drivers/pcmcia/ds.c*	Registers a driver with the PCMCIA core
`pcmcia_unregister_driver()`	*drivers/pcmcia/ds.c*	Unregisters a driver from the PCMCIA core

`pcmcia_get_first_tuple()`
`pcmcia_get_tuple_data()`
`pcmcia_parse_tuple()`
	include/pcmcia/cistpl.h
	drivers/pcmcia/cistpl.c
		Library routines to manipulate CIS space

`pcmcia_request_irq()`	*drivers/pcmcia/pcmcia_resource.c*	Gets an IRQ assigned for a PCMCIA card

Chapter 10. Peripheral Component Interconnect

In This Chapter

- The PCI Family 278
- Addressing and Identification 281
- Accessing PCI Regions 285
- Direct Memory Access 288
- Device Example: Ethernet-Modem Card 292
- Debugging 308
- Looking at the Sources 308

Peripheral Component Interconnect (PCI) is an omnipresent I/O backbone. Whether you are backing up data on a storage server, capturing video from your desktop, or surfing the web from your laptop, PCI might be serving you in some avatar or the other. PCI, and form factors adapted or derived from PCI such as Mini PCI, CardBus, PCI Extended, PCI Express, PCI Express Mini Card, and Express Card have become de facto peripheral connection technologies on today's computers.

The PCI Family

PCI is a high-speed bus used for communication between the CPU and I/O devices. The PCI specification enables transfer of 32 bits of data in parallel at 33MHz or 66MHz, yielding a peak throughput of 266MBps.

CardBus is a derivative of PCI and has the form factor of a PC Card. CardBus cards are also 32-bits wide and run at 33MHz. Even though CardBus and PCMCIA cards use the same 68-pin connectors, CardBus devices support 32 data lines compared to 16 for PCMCIA by multiplexing address and data lines as done in the PCI bus.

Mini PCI, also a 33MHz 32-bit bus, is another adaptation of PCI found in small-footprint computers such as laptops. A PCI card can talk via a Mini PCI slot using a compatible connector.

An extension to PCI called PCI Extended (or PCI-X) expands the bus width to 64 bits, frequency to 133MHz, and the throughput to about 1GBps. PCI-X 2.0 is the current version of the standard.

PCI Express (PCIe or PCI-E) is the present generation of the PCI family. Unlike the parallel PCI bus, PCIe uses a serial protocol to transfer data. PCIe supports a maximum of 32 serial *links.* Each PCIe link (in the commonly used version 1.1 of the specification) yields a throughput of 250MBps in each transfer direction, thus producing a maximum PCIe data rate of 8GBps in each direction. PCIe 2.0 is the current version of the standard and supports higher data rates.

Serial communication is faster and cheaper than parallel data transfer due to the absence of factors such as signal interference, so the industry trend is to move from parallel buses to serial technologies. PCIe and its adaptations aim to replace PCI and its derivatives, and this shift is also part of the methodology change from parallel to serial communication. Several I/O interfaces discussed in this book, such as RS-232, USB, FireWire, SATA, Ethernet, Fibre Channel, and InfiniBand, are serial communication architectures.

The CardBus equivalent in the PCIe family is the Express Card. Express Cards directly connect to the system bus via a PCIe link or USB 2.0 (discussed in the next chapter), and circumvent middlemen such as CardBus controllers. Mini PCI's cousin in the PCIe family is PCI Express Mini Card.

Recent laptops support Express Card slots instead of (or in addition to) CardBus, and PCI Express Mini Card slots in place of Mini PCI. The former two have smaller footprints and higher speeds compared to the latter two.

Table 10.1 summarizes the important relatives of PCI. From the kernel's perspective, all these technologies are compatible with one another. A kernel PCI driver will work with all related technologies mentioned previously; so even though we base example code in this chapter on a CardBus card, the concepts apply to other PCI derivatives, too.

Table 10.1. PCI's Siblings, Children, and Cousins

Bus Name	Characteristics	Form Factor
PCI	32-bit bus at 33MHz or 66MHz; yields up to 266MBps.	Internal slot in desktops and servers.
Mini PCI	32-bit bus at 33MHz.	Internal slot in laptops.
CardBus	32-bit bus at 33MHz.	External PC card slot in laptops. Compatible with PCI.
PCI Extended (PCI-X)	64-bit bus at 133 MHz, yielding up to 1GBps.	Internal slot in desktops and servers. Wider than PCI, but a PCI card can be plugged into a PCI-X slot.
PCI Express (PCIe)	250MBps per PCIe link in each transfer direction, yielding a maximum throughput of 8GBps in each direction.	Replaces the internal PCI slot in newer systems. PCIe is a serial protocol unlike native PCI, which is parallel.
PCI Express Mini Card	250MBps in each direction if the interface is based on a PCIe link; 60MBps if the interface is based on USB 2.0.	Replaces Mini PCI as the internal slot in newer laptops. Smaller form factor than Mini PCI.

Bus Name	Characteristics	Form Factor
Express Card	250MBps in each direction if the interface is based on a PCIe link; 60MBps if the interface is based on USB 2.0.	Thin external slot in newer laptops that replaces CardBus. Interfaces with the system bus via PCIe or USB 2.0.

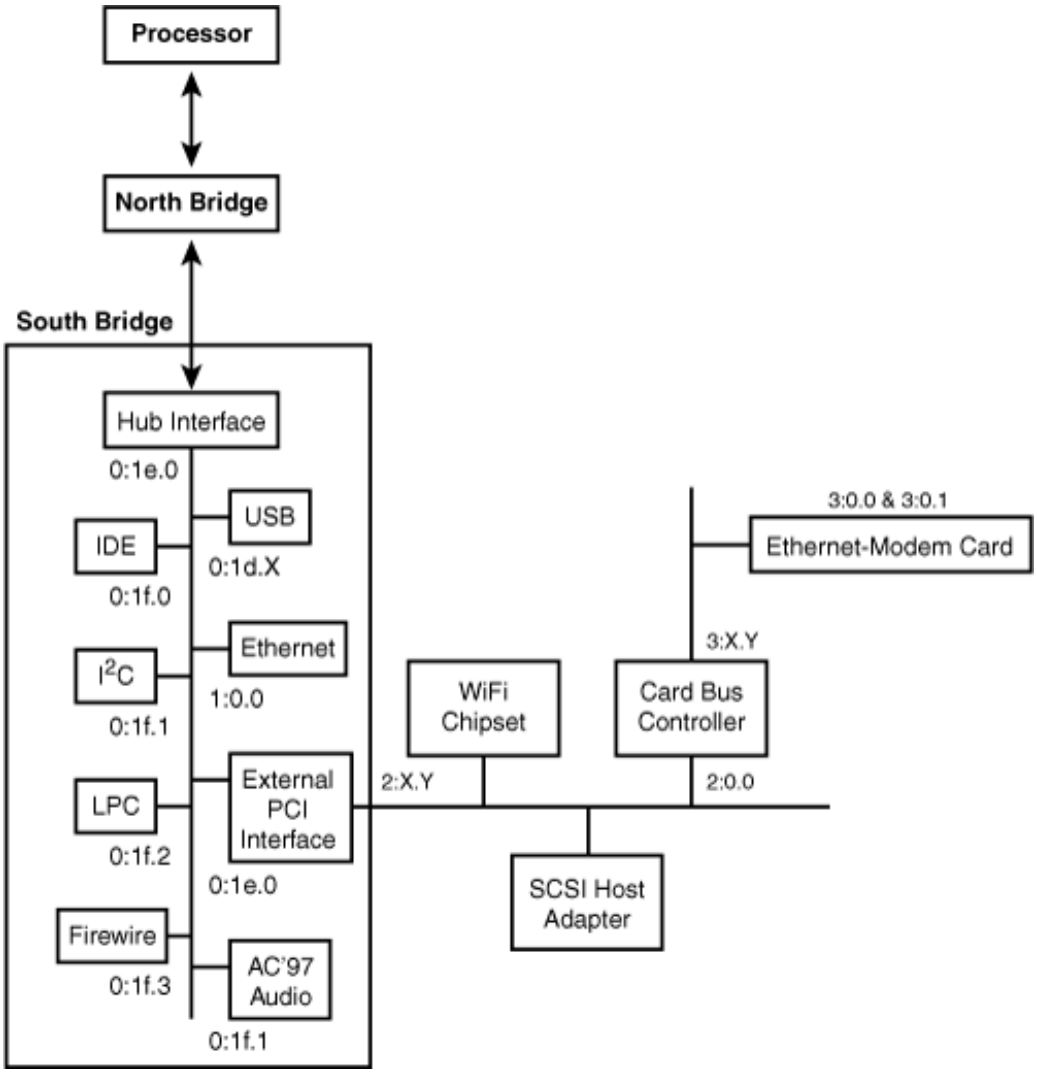
Solutions based on the PCI family are available for a vast spectrum of hardware domains:

- Networking technologies such as Gigabit Ethernet, WiFi, ATM, Token Ring, and ISDN.
- Host adapters for storage technologies, such as SCSI.
- Host controllers for I/O buses such as USB, FireWire, IDE, I^2C, and PCMCIA. On PC-compatible systems, these host controllers function as bridges between the PCI controller on the South Bridge and the bus technology they source. Verify this by running *lspci* (discussed later).
- Graphics, video streaming, and data capture.
- Serial port and parallel port cards.
- Sound cards.
- Devices such as Watchdogs, EDAC-capable memory controllers, and game ports.

For the driver developer, the PCI family offers an attractive advantage: a system of automatic device configuration. Unlike drivers for the older ISA generation, PCI drivers need not implement complex probing logic. During boot, the BIOS-type boot firmware (or the kernel itself if so configured) walks the PCI bus and assigns resources such as interrupt levels and I/O base addresses. The device driver gleans this assignment by peeking at a memory region called the PCI configuration space. PCI devices possess 256 bytes of configuration memory. The top 64 bytes of the configuration space is standardized and holds registers that contain details such as the status, interrupt line, and I/O base addresses. PCIe and PCI-X 2.0 offer an extended configuration space of 4KB. We will learn how to operate on the PCI configuration space later.

Figure 10.1 shows PCI in a PC-compatible system. Components integrated into the South Bridge such as controller silicon for USB, IDE, I^2C, LPC, and Ethernet reside off the PCI bus. Some of these controllers contain an internal PCI-to-PCI bridge to source a dedicated PCI bus for the respective I/O technology. The South Bridge additionally contains an external PCI bus to connect I/O peripherals such as CardBus controllers and WiFi chipsets. Figure 10.1 also shows PCI address tuples corresponding to each connected subsystem. This will get clearer when we learn about PCI addressing next.

Figure 10.1. PCI inside a PC South Bridge.


Processor
North Bridge
South Bridge
Hub Interface
0:1e.0
USB
0:1d.X
IDE
0:1f.0
Ethernet
1:0.0
I2C
0:1f.1
LPC
0:1f.2
External PCI Interface
0:1e.0
Firewire
0:1f.3
AC'97 Audio
0:1f.1
WiFi Chipset
2:X.Y
Card Bus Controller
2:0.0
3:X.Y
3:0.0 & 3:0.1
Ethernet-Modem Card
SCSI Host Adapter

Chapter 10. Peripheral Component Interconnect

In This Chapter

- The PCI Family 278
- Addressing and Identification 281
- Accessing PCI Regions 285
- Direct Memory Access 288
- Device Example: Ethernet-Modem Card 292
- Debugging 308
- Looking at the Sources 308

Peripheral Component Interconnect (PCI) is an omnipresent I/O backbone. Whether you are backing up data on a storage server, capturing video from your desktop, or surfing the web from your laptop, PCI might be serving you in some avatar or the other. PCI, and form factors adapted or derived from PCI such as Mini PCI, CardBus, PCI Extended, PCI Express, PCI Express Mini Card, and Express Card have become de facto peripheral connection technologies on today's computers.

The PCI Family

PCI is a high-speed bus used for communication between the CPU and I/O devices. The PCI specification enables transfer of 32 bits of data in parallel at 33MHz or 66MHz, yielding a peak throughput of 266MBps.

CardBus is a derivative of PCI and has the form factor of a PC Card. CardBus cards are also 32-bits wide and run at 33MHz. Even though CardBus and PCMCIA cards use the same 68-pin connectors, CardBus devices support 32 data lines compared to 16 for PCMCIA by multiplexing address and data lines as done in the PCI bus.

Mini PCI, also a 33MHz 32-bit bus, is another adaptation of PCI found in small-footprint computers such as laptops. A PCI card can talk via a Mini PCI slot using a compatible connector.

An extension to PCI called PCI Extended (or PCI-X) expands the bus width to 64 bits, frequency to 133MHz, and the throughput to about 1GBps. PCI-X 2.0 is the current version of the standard.

PCI Express (PCIe or PCI-E) is the present generation of the PCI family. Unlike the parallel PCI bus, PCIe uses a serial protocol to transfer data. PCIe supports a maximum of 32 serial *links.* Each PCIe link (in the commonly used version 1.1 of the specification) yields a throughput of 250MBps in each transfer direction, thus producing a maximum PCIe data rate of 8GBps in each direction. PCIe 2.0 is the current version of the standard and supports higher data rates.

Serial communication is faster and cheaper than parallel data transfer due to the absence of factors such as signal interference, so the industry trend is to move from parallel buses to serial technologies. PCIe and its adaptations aim to replace PCI and its derivatives, and this shift is also part of the methodology change from parallel to serial communication. Several I/O interfaces discussed in this book, such as RS-232, USB, FireWire, SATA, Ethernet, Fibre Channel, and InfiniBand, are serial communication architectures.

The CardBus equivalent in the PCIe family is the Express Card. Express Cards directly connect to the system bus via a PCIe link or USB 2.0 (discussed in the next chapter), and circumvent middlemen such as CardBus controllers. Mini PCI's cousin in the PCIe family is PCI Express Mini Card.

Recent laptops support Express Card slots instead of (or in addition to) CardBus, and PCI Express Mini Card slots in place of Mini PCI. The former two have smaller footprints and higher speeds compared to the latter two.

Table 10.1 summarizes the important relatives of PCI. From the kernel's perspective, all these technologies are compatible with one another. A kernel PCI driver will work with all related technologies mentioned previously; so even though we base example code in this chapter on a CardBus card, the concepts apply to other PCI derivatives, too.

Table 10.1. PCI's Siblings, Children, and Cousins

Bus Name	Characteristics	Form Factor
PCI	32-bit bus at 33MHz or 66MHz; yields up to 266MBps.	Internal slot in desktops and servers.
Mini PCI	32-bit bus at 33MHz.	Internal slot in laptops.
CardBus	32-bit bus at 33MHz.	External PC card slot in laptops. Compatible with PCI.
PCI Extended (PCI-X)	64-bit bus at 133 MHz, yielding up to 1GBps.	Internal slot in desktops and servers. Wider than PCI, but a PCI card can be plugged into a PCI-X slot.
PCI Express (PCIe)	250MBps per PCIe link in each transfer direction, yielding a maximum throughput of 8GBps in each direction.	Replaces the internal PCI slot in newer systems. PCIe is a serial protocol unlike native PCI, which is parallel.
PCI Express Mini Card	250MBps in each direction if the interface is based on a PCIe link; 60MBps if the interface is based on USB 2.0.	Replaces Mini PCI as the internal slot in newer laptops. Smaller form factor than Mini PCI.

Bus Name	Characteristics	Form Factor
Express Card	250MBps in each direction if the interface is based on a PCIe link; 60MBps if the interface is based on USB 2.0.	Thin external slot in newer laptops that replaces CardBus. Interfaces with the system bus via PCIe or USB 2.0.

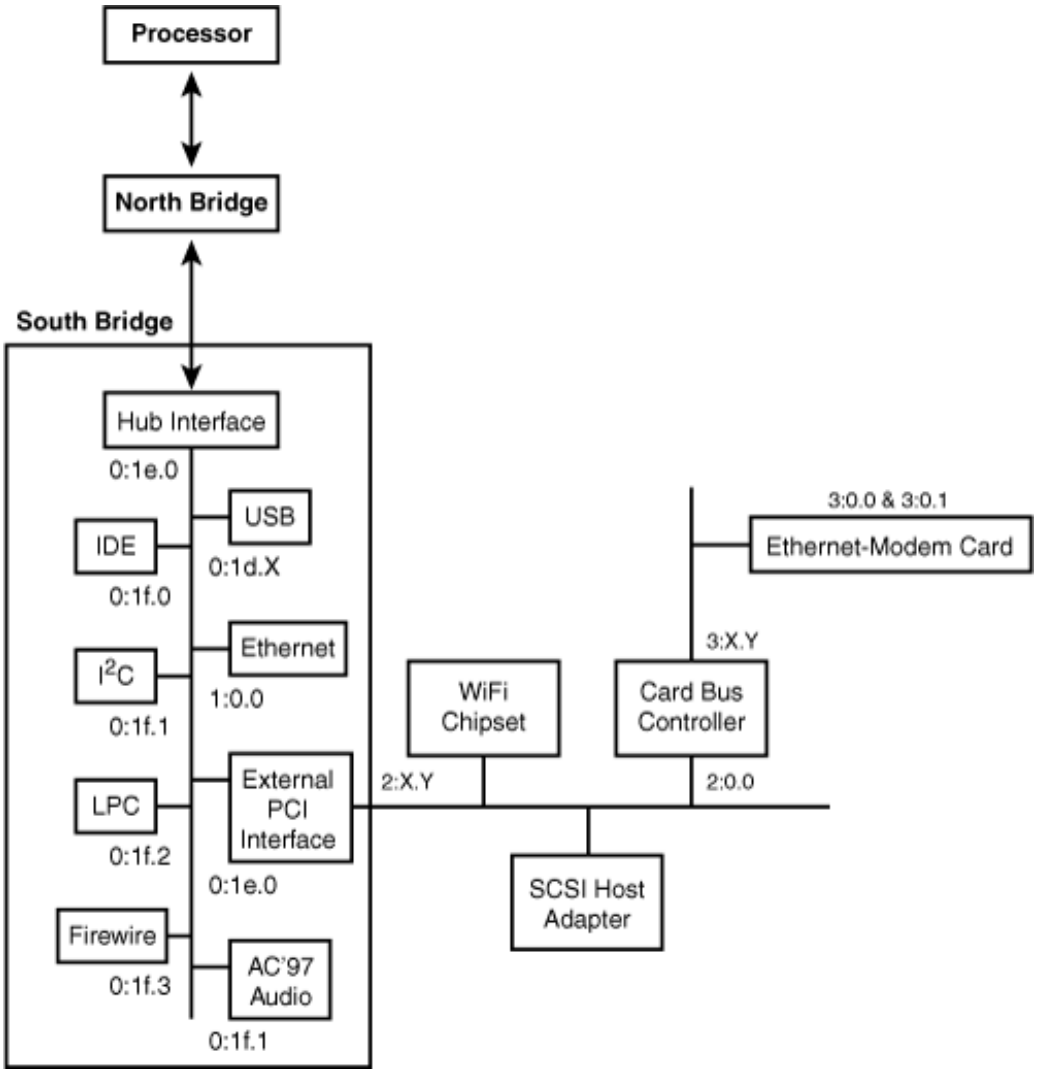
Solutions based on the PCI family are available for a vast spectrum of hardware domains:

- Networking technologies such as Gigabit Ethernet, WiFi, ATM, Token Ring, and ISDN.
- Host adapters for storage technologies, such as SCSI.
- Host controllers for I/O buses such as USB, FireWire, IDE, I^2C, and PCMCIA. On PC-compatible systems, these host controllers function as bridges between the PCI controller on the South Bridge and the bus technology they source. Verify this by running *lspci* (discussed later).
- Graphics, video streaming, and data capture.
- Serial port and parallel port cards.
- Sound cards.
- Devices such as Watchdogs, EDAC-capable memory controllers, and game ports.

For the driver developer, the PCI family offers an attractive advantage: a system of automatic device configuration. Unlike drivers for the older ISA generation, PCI drivers need not implement complex probing logic. During boot, the BIOS-type boot firmware (or the kernel itself if so configured) walks the PCI bus and assigns resources such as interrupt levels and I/O base addresses. The device driver gleans this assignment by peeking at a memory region called the PCI configuration space. PCI devices possess 256 bytes of configuration memory. The top 64 bytes of the configuration space is standardized and holds registers that contain details such as the status, interrupt line, and I/O base addresses. PCIe and PCI-X 2.0 offer an extended configuration space of 4KB. We will learn how to operate on the PCI configuration space later.

Figure 10.1 shows PCI in a PC-compatible system. Components integrated into the South Bridge such as controller silicon for USB, IDE, I^2C, LPC, and Ethernet reside off the PCI bus. Some of these controllers contain an internal PCI-to-PCI bridge to source a dedicated PCI bus for the respective I/O technology. The South Bridge additionally contains an external PCI bus to connect I/O peripherals such as CardBus controllers and WiFi chipsets. Figure 10.1 also shows PCI address tuples corresponding to each connected subsystem. This will get clearer when we learn about PCI addressing next.

Figure 10.1. PCI inside a PC South Bridge.


Processor
North Bridge
South Bridge
Hub Interface
0:1e.0
USB
0:1d.X
IDE
0:1f.0
Ethernet
1:0.0
I2C
0:1f.1
LPC
0:1f.2
External PCI Interface
0:1e.0
Firewire
0:1f.3
AC'97 Audio
0:1f.1
WiFi Chipset
2:X.Y
Card Bus Controller
2:0.0
3:X.Y
3:0.0 & 3:0.1
Ethernet-Modem Card
SCSI Host Adapter

Addressing and Identification

PCI devices are addressed using *bus*, *device*, and *function* numbers, and they are identified via *vendorIDs*, *deviceIDs*, and *class codes*. Let's learn these concepts with the help of the *lspci* utility that is part of the *PCI Utilities* package downloadable from http://mj.ucw.cz/pciutils.shtml.

Assume that you're using a Xircom Ethernet-Modem multifunction CardBus card on a Pentium-class laptop served by a Texas Instruments PCI4510 CardBus controller, as shown in Figure 10.1. Run lspci:

Code View:
```
bash>lspci
00:00.0 Host bridge: Intel Corporation 82852/82855 GM/GME/PM/GMV Processor to I/O
Controller (rev 02)
...
02:00.0 CardBus bridge: Texas Instruments PCI4510 PC card Cardbus Controller (rev 03)
...
03:00.0 Ethernet controller: Xircom Cardbus Ethernet 10/100 (rev 03)
03:00.1 Serial controller: Xircom Cardbus Ethernet + 56k Modem (rev 03)
```

Consider the tuple `(XX:YY.Z)` at the beginning of each entry in the preceding output. `XX` stands for the PCI bus number. A PCI domain can host up to `256` buses. In the laptop used previously, the CardBus bridge is connected to PCI bus `2`. This bridge sources another PCI bus numbered `3` that hosts the Xircom card.

`YY` is the PCI device number. Each bus can connect to a maximum of 32 PCI devices. Each device can, in turn, implement up to eight functions represented by `Z`. The Xircom card can simultaneously perform two functions. Thus, `03:00.0` addresses the Ethernet function of the card, while `03:00.1` corresponds to its modem communication function. Issue `lspci –t` to elicit a tree-like layout of the PCI buses and devices on your system:

```
bash> lspci –t
-[0000:00]-+-00.0
           +-00.1
           +-00.3
           +-02.0
           +-02.1
           +-1d.0
           +-1d.1
           +-1d.2
           +-1d.7
           +-1e.0-[0000:02-05]--+-[0000:03]-+-00.0
           |                    |           \-00.1
           |                    \-[0000:02]-+-00.0
           |                                +-00.1
           |                                +-01.0
           |                                \-02.0
           +-1f.0
```

As you can see from the preceding output (and in Figure 10.1), to walk the PCI bus and reach the Xircom modem (`03:00.01`) or Ethernet controller (`03:00.0`), you have to start from your PCI domain (labeled `0000` in

the preceding output), traverse a PCI-to-PCI bridge (00:1e.0), and then cross a PCI-to-CardBus host controller (02:0.0). The sysfs representation of the PCI subsystem mirrors this layout:

```
bash> ls /sys/devices/pci0000:00/0000:00:1e.0/0000:02:00.0/0000:03:00.0/
...
net:eth2  ⟶ Ethernet
...
bash> ls /sys/devices/pci0000:00/0000:00:1e.0/0000:02:00.0/0000:03:00.1/
...
tty:ttyS1 ⟶ Modem
...
```

As you saw earlier, PCI devices possess a 256-byte memory region that holds configuration registers. This space is the key to identify the make and capabilities of PCI cards. Let's take a peek inside the configuration spaces of the CardBus controller and the Xircom dual-function card previously used. The Xircom card has two configuration spaces, one per supported function:

Code View:

```
bash> lspci –x
00:00.0 Host bridge: Intel Corporation 82852/82855 GM/GME/PM/GMV Processor to I/O
Controller (rev 02)
00: 86 80 80 35 06 01 90 20 02 00 00 06 00 00 80 00
10: 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 14 10 5c 05
30: 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00
...
02:00.0 CardBus bridge: Texas Instruments PCI4510 PC card Cardbus Controller (rev 03)
00: 4c 10 44 ac 07 00 10 02 03 00 07 06 20 a8 82 00
10: 00 00 00 b0 a0 00 00 22 02 03 04 b0 00 00 00 f0
20: 00 f0 ff f1 00 00 00 d2 00 f0 ff d3 00 30 00 00
30: fc 30 00 00 00 34 00 00 fc 34 00 00 0b 01 00 05
...
03:00.0 Ethernet controller: Xircom Cardbus Ethernet 10/100 (rev 03)
00: 5d 11 03 00 07 00 10 02 03 00 00 02 00 40 80 00
10: 01 30 00 00 00 00 00 d2 00 08 00 d2 00 00 00 00
20: 00 00 00 00 00 00 00 00 07 01 00 00 5d 11 81 11
30: 00 00 00 00 dc 00 00 00 00 00 00 00 0b 01 14 28
03:00.1 Serial controller: Xircom Cardbus Ethernet + 56k Modem (rev 03)
00: 5d 11 03 01 03 00 10 02 03 02 00 07 00 00 80 00
10: 81 30 00 00 00 10 00 d2 00 18 00 d2 00 00 00 00
20: 00 00 00 00 00 00 00 00 07 02 00 00 5d 11 81 11
30: 00 00 00 00 dc 00 00 00 00 00 00 00 0b 01 00 00
```

PCI registers are little-endian, so factor that while interpreting the preceding output. You may also dump PCI configuration regions via sysfs. So, to look at the configuration space of the Ethernet function of the Xircom card, do this:

Code View:

```
bash> od -x /sys/devices/pci0000:00/0000:00:1e.0/0000:02:00.0/0000:03:00.1/config
0000000 115d 0003 0007 0210 0003 0200 4000 0080
0000020 3001 0000 0000 d200 0800 d200 0000 0000
0000040 0000 0000 0000 0000 0107 0000 115d 1181
```

...

Table 10.2 explains some of the values shown in the preceding dump. The first two bytes contain the vendor ID, which identifies the company that manufactured the card. PCI vendor IDs are maintained and assigned globally. (Point your browser to www.pcidatabase.com for a database.) As you can decipher from the preceding output, Intel, Texas Instruments, and Xircom (now acquired by Intel) own vendor IDs of `0x8086`, `0x104C`, and `0x115D`, respectively. The next two bytes are specific to the functionality of the card and constitute its device ID. From the preceding output, the Ethernet functionality of the Xircom card owns a device ID of `0x0003`, while the modem answers to a device ID of `0x0103`. PCI cards additionally possess subvendor and subdevice IDs (see words at offsets 44 and 46 in the preceding dump) to further pinpoint their identity.

Table 10.2. PCI Configuration Space Semantics

Configuration Space Offset	Semantics	Values from the Dump Output for the Xircom Card
0	Vendor ID	0x115D
2	Device ID	0x0003
10	Class code	0x0200
16 to 39	Base address register 0 (BAR 0) to BAR5	0x3001...0000
44	Subvendor ID	0x115D
46	Subdevice ID	0x1181

Ten bytes into the configuration space lies the code that describes the class of the device. PCI bridges have a class code starting with `0x06`, network devices possess a class code beginning with `0x02`, and communication devices own a class code commencing with `0x07`. Thus, in the preceding example, the CardBus bridge, the Ethernet card, and the serial modem own class codes of `0x0607`, `0x0200`, and `0x0700`, respectively. You can find class code definitions in *include/linux/pci_ids.h*.

PCI drivers register the vendor IDs, device IDs, and class codes that they support with the PCI subsystem. Using this database, the PCI subsystem binds an inserted card to the appropriate device driver after gleaning its identity from its configuration space. We will see how this is done when we implement an example driver later.

Accessing PCI Regions

PCI devices contain three addressable regions: configuration space, I/O ports, and device memory. Let's learn how to access these memory regions from a device driver.

Configuration Space

The kernel offers a set of six functions that your driver can use to operate on PCI configuration space:

```
pci_read_config_[byte|word|dword](struct pci_dev *pdev,
                                  int offset, int *value);
and
pci_write_config_[byte|word|dword](struct pci_dev *pdev,
                                   int offset, int value);
```

In the argument list, `struct pci_dev` is the PCI device structure, and `offset` is the byte position in the configuration space that you want to access. For read functions, `value` is a pointer to a supplied data buffer, and for write routines, it contains the data to be written.

Let's consider some examples:

- To decipher the IRQ number assigned to a card function, use the following:

  ```
  unsigned char irq;
  pci_read_config_byte(pdev, PCI_INTERRUPT_LINE, &irq);
  ```

  As per the PCI specification, offset `60` inside the PCI configuration space holds the IRQ number assigned to the card. All configuration register offsets are expressively defined in *include/linux/pci_regs.h*, so use `PCI_INTERRUPT_LINE` rather than `60` to specify this offset. Similarly, to read the PCI status register (two bytes at offset six in the configuration space), do this:

  ```
  unsigned short status;
  pci_read_config_word(pdev, PCI_STATUS, &status);
  ```

- Only the first 64 bytes of the configuration space are standardized. The device manufacturer defines desired semantics to the rest. The Xircom card used earlier, assigns four bytes at offset 64 for power management purposes. To disable power management, the Xircom CardBus driver, *drivers/net/tulip/xircom_cb.c*, does this:

  ```
  #define PCI_POWERMGMT 0x40
  pci_write_config_dword(pdev, PCI_POWERMGMT, 0x0000);
  ```

I/O and Memory

PCI cards have up to six I/O or memory regions. I/O regions contain registers, and memory regions hold data. Video cards, for example, have I/O spaces that accommodate control registers and memory regions that map to frame buffers. Not all cards have addressable memory regions, however. The semantics of I/O and memory spaces are hardware-dependent and can be obtained from the device data sheet.

Like for configuration memory, the kernel offers a set of helpers to operate on I/O and memory regions of PCI devices:

Code View:
```
unsigned long pci_resource_[start|len|end|flags] (struct pci_dev *pdev, int bar);
```

To operate on an I/O region such as the device control registers of a PCI video card, the driver needs to do the following:

1. Get the I/O base address from the appropriate base address register (`bar`) in the configuration space:

   ```
   unsigned long io_base = pci_resource_start(pdev, bar);
   ```

   This assumes that the device control registers for this card are mapped to the memory region associated with `bar`, whose value can range from 0 through 5, as shown in Table 10.2.

2. Mark this region as being spoken for, using the kernel's `request_region()` regulatory mechanism discussed in Chapter 5, "Character Drivers":

   ```
   request_region(io_base, length, "my_driver");
   ```

   Here, `length` is the size of the control register space and `my_driver` identifies the region's owner. Look for the entry containing `my_driver` in */proc/ioports* to spot this memory region.

   You may instead use the wrapper function `pci_request_region()`, defined in *drivers/pci/pci.c*.

3. Add the register's offset obtained from the data-sheet, to the base address gleaned in Step 1. Operate on this address using the `inb()` and `outb()` family of functions discussed in Chapter 5:

   ```
   /* Read */
   register_data = inl(io_base + REGISTER_OFFSET);
   /* Use */
   /* ... */
   /* Write */
   outl(register_data, iobase + REGISTER_OFFSET);
   ```

To operate on a memory region such as the frame buffer on the above PCI video card, follow these steps:

1. Get the base address, length, and flags associated with the memory region:

   ```
   unsigned long mmio_base   = pci_resource_start(pdev, bar);
   unsigned long mmio_length = pci_resource_length(pdev, bar);
   unsigned long mmio_flags  = pci_resource_flags(pdev, bar);
   ```

   This assumes that this memory is mapped to the base address register, `bar`.

2. Mark ownership of this region using the kernel's `request_mem_region()` regulatory mechanism:

   ```
   request_mem_region(mmio_base, mmio_length, "my_driver");
   ```

   You may instead use the wrapper function `pci_request_region()`, mentioned previously.

3. Obtain CPU access to the device memory obtained in Step 1. Certain memory regions, such as the ones that hold registers, need to guard against side effects, so they are marked as not being prefetchable (or cacheable) by the CPU. Other regions, such as the one used in this example, can be cached. Depending on the access flag, use the appropriate function to obtain kernel virtual addresses corresponding to the mapped region:

   ```
   void __iomem *buffer;
   if (flags & IORESOURCE_CACHEABLE) {
     buffer = ioremap(mmio_base, mmio_length);
   } else {
     buffer = ioremap_nocache(mmio_base, mmio_length);
   }
   ```

To be safe, and to avoid performing the preceding checks, use the services of `pci_iomap()` defined in *lib/iomap.c* instead:

```
buffer = pci_iomap(pdev, bar, mmio_length);
```

Direct Memory Access

Direct Memory Access (DMA) is the capability to transfer data from a peripheral to main memory without the CPU's intervention. DMA boosts the performance of peripherals manyfold, because it doesn't burn CPU cycles to move data. PCI networking cards and IDE disk drives are common examples of peripherals relying on DMA for data transfer.

DMA is initiated by a DMA master. The PC motherboard has a DMA controller on the South Bridge that can master the I/O bus and initiate DMA to or from a peripheral. This is usually the case for legacy ISA cards. However, buses such as PCI can master the bus and initiate DMA transfers. CardBus cards are similar to PCI and also support DMA mastering. PCMCIA devices, on the other hand, do not support DMA mastering, but the PCMCIA controller, which is usually wired to a PCI bus, might have DMA mastering capabilities.

The issue of *cache coherency* is synonymous with DMA. For optimum performance, processors cache recently accessed bytes, so data passing between the CPU and main memory streams through the processor cache. During DMA, however, data travels directly between the DMA controller and main memory and, hence, bypasses the processor cache. This evasion has the potential to introduce inconsistencies because the processor might work on stale data living in its cache. Some architectures automatically synchronize the cache with main memory using a technique called *bus snooping.* Many others rely on software to achieve coherency, however. We will learn how to perform coherent DMA operations after introducing a few more topics.

DMA can occur *synchronously* or *asynchronously.* An example of the former is DMA from a system frame buffer to an LCD controller. A user application writes pixel data to a DMA-mapped frame buffer via */dev/fbX*, while the LCD controller uses DMA to collect this data synchronously at timed intervals. We discuss more about this in Chapter 12, "Video Drivers." An example of asynchronous DMA is the transmit and receive of data frames between the CPU and a network card discussed in Chapter 15, "Network Interface Cards."

System memory regions that are the source or destination of DMA transfers are called DMA buffers. If a bus interface has addressing limitations, that'll affect the memory range that can hold DMA buffers. So, DMA buffers suitable for a 24-bit bus such as ISA can live only in the bottom 16MB of system memory called `ZONE_DMA` (see the section "Allocating Memory" in Chapter 2, "A Peek Inside the Kernel"). PCI buses are 32-bits wide by default, so you won't usually face such limitations on 32-bit platforms. To inform the kernel about any special needs of DMA-able buffers, use the following:

```
dma_set_mask(struct device *dev, u64 mask);
```

If this function returns success, you may DMA to any address that is `mask` bits in length. For example, the e1000 PCI-X Gigabit Ethernet driver (*drivers/net/e1000/e1000_main.c*) does the following:

```
if (!(err = pci_set_dma_mask(pdev, DMA_64BIT_MASK))) {
   /* System supports 64-bit DMA */
   pci_using_dac = 1;
} else {
  /* See if 32-bit DMA is supported */
  if ((err = pci_set_dma_mask(pdev, DMA_32BIT_MASK))) {
    /* No, let's abort */
    E1000_ERR("No usable DMA configuration, aborting\n");
    return err;
  }
  /* 32-bit DMA */
  pci_using_dac = 0;
}
```

I/O devices view DMA buffers through the lens of the bus controller and any intervening *I/O memory management unit* (IOMMU). Because of this, I/O devices work with *bus* addresses, rather than physical or kernel virtual addresses. So, when you inform a PCI card about the location of a DMA buffer, you have to let it know the buffer's bus address. DMA service routines map the kernel virtual address of DMA buffers to bus addresses so that both the device and the CPU can access the buffers. Bus addresses are of type `dma_addr_t`, defined in *include/asm-your-arch/types.h*.

There are a couple more concepts worth knowing about DMA. One is the idea of *bounce buffers.* Bounce buffers reside in DMA-able regions and are used as temporary memory when DMA is requested to/from non-DMA-able memory regions. An example is DMA to an address higher than 4GB from a 32-bit PCI peripheral when there is no intervening IOMMU. Data is first transferred to a bounce buffer and then copied to the final destination. The second concept is a flavor of DMA called *scatter-gather.* When data to be DMA'ed is spread over discontinuous regions, scatter-gather capability enables the hardware to gather contents of the scattered buffers at one go. The reverse occurs when data is DMA'ed from the card to buffers scattered in memory. Scatter-gather capability boosts performance by eliminating the need to service multiple DMA requests.

The kernel features a healthy API that masks many of the internal details of configuring DMA. This API gets simpler if you are writing a driver for a PCI card that supports bus mastering. (Most PCI cards do.) PCI DMA routines are essentially wrappers around the generic DMA service routines and are defined in *include/asm-generic/pci-dma-compat.h.* In this chapter, we use only the PCI DMA API.

The kernel provides two classes of DMA service routines to PCI drivers:

1. *Consistent* (or coherent) DMA access methods. These routines guarantee data coherency in the face of DMA activity. If both the PCI device and the CPU are likely to frequently operate on a DMA buffer, consistency is crucial, so use the consistent API. The trade-off is a degree of performance penalty. To obtain a consistent DMA buffer, call this service routine:

   ```
   void *pci_alloc_consistent(struct pci_dev *pdev,
                              size_t size,
                              dma_addr_t *dma_handle);
   ```

   This function allocates a DMA buffer, generates its bus address, and returns the associated kernel virtual address. The first two arguments respectively hold the PCI device structure (which is discussed later) and the size of the requested DMA buffer. The third argument, `dma_handle`, is a pointer to the bus address that the function call generates. The following snippet allocates and frees a consistent DMA buffer:

   ```
   /* Allocate */
   void *vaddr = pci_alloc_consistent(pdev, size,
                                      &dma_handle);
   /* Use */
   /* ... */
   /* Free */
   pci_free_consistent(pdev, size, vaddr, dma_handle);
   ```

2. *Streaming* DMA access methods. These routines do not guarantee consistency and are faster as a result. They are useful when there is not much need for shared access between the CPU and the I/O device. When a streamed buffer has been mapped for device access, the driver has to explicitly unmap (or sync) it before the CPU can reliably operate on it. There are two families of streaming access routines: `pci_[map|unmap|dma_sync]_single()` and `pci_[map|unmap|dma_sync]_sg()`.

   The first function family maps, unmaps, and synchronizes a single preallocated DMA buffer. `pci_map_single()` is prototyped as follows:

   ```
   dma_addr_t pci_map_single(struct pci_dev *pdev, void *ptr,
   ```

```
                                size_t size, int direction);
```

The first three arguments respectively hold the PCI device structure, the kernel virtual address of a preallocated DMA buffer, and the size of the supplied buffer. The fourth argument, `direction`, can be one of the following: `PCI_DMA_BIDIRECTION`, `PCI_DMA_TODEVICE`, `PCI_DMA_FROMDEVICE`, or `PCI_DMA_NONE`. The names are self-explanatory, but the first option is expensive, and the last is for debugging. We discuss streamed DMA mapping further when we develop an example driver later.

The second function family maps, unmaps, and synchronizes a scatter-gather list of DMA buffers. `pci_map_sg()` is prototyped as follows:

```
int pci_map_sg(struct pci_dev *pdev,
               struct scatterlist *sgl,
               int num_entries, int direction);
```

The scattered list is specified using the second argument, `struct scatterlist`, defined in *include/asm-your-arch/scatterlist.h.* `num_entries` is the number of entries in the `scatterlist`. The first and last arguments are the same as that described for `pci_map_single()`. The function returns the number of mapped entries:

```
num_mapped = pci_map_sg(pdev, sgl, num_entries,
                        PCI_DMA_TODEVICE);
for (i=0; i<num_mapped; i++) {
  /* sg_dma_address(&sgl[i]) returns the bus address
     of this entry */
  /* sg_dma_len(&sgl[i]) returns the length of this region
   */
}
```

Let's summarize the characteristics of coherent and streaming DMA to help you decide their suitability for your usage scenario:

- Coherent mappings are simple to code but expensive to use. Streaming mappings have the reverse characteristic.

- Coherent mappings are preferred when both the CPU and the I/O device need to frequently manipulate the DMA buffer. This is usually the case for synchronous DMA. An example is the frame buffer driver mentioned previously, where each DMA operates on the same buffer. Because the CPU and the video controller are constantly accessing the frame buffer, it makes sense to use coherent mappings in this situation.

- Use streaming mappings when the I/O device owns the buffer for long durations. Streamed DMA is common for asynchronous operation when each DMA operates on a different buffer. An example is a network driver, where the buffers that carry transmit packets are mapped and unmapped on-the-fly.

- DMA descriptors are good candidates for coherent mapping. DMA descriptors contain metadata about DMA buffers such as their address and length and are frequently accessed by both the CPU and the device. Mapping descriptors on-the-fly is expensive because that entails frequent unmappings and remappings (or sync operations).

Device Example: Ethernet-Modem Card

Armed with the knowledge acquired so far, let's venture to write a skeletal device driver for a fictitious Ethernet-Modem dual-function CardBus card and see how it can be used for networking on a LAN and for establishing a dialup connection to an Internet service provider. You will essentially need one device driver per supported function. Assuming you already have a serial driver (we learned to write serial drivers in Chapter 6, "Serial Drivers") and an Ethernet driver (we will learn to implement network drivers in Chapter 15) that support the chipsets used on the card, let's tinker with those drivers and get them to work with the CardBus interface. The example here is generic but is loosely based on the kernel driver for the Xircom card that we used previously. The Ethernet and modem portions of the Xircom driver live separately in *drivers/net/tulip/xircom_cb.c* and *drivers/serial/8250_pci.c*, respectively.

Initializing and Probing

PCI drivers use an array of `pci_device_id` structures defined in *include/linux/mod_devicetable.h* to describe the identity of the cards they support:

```
struct pci_device_id {
  __u32 vendor, device;         /* Vendor and Device IDs */
  __u32 subvendor, subdevice;   /* Subvendor and Subdevice IDs */
  __u32 class, classmask;       /* Class and class mask */
  kernel_ulong_t driver_data;   /* Private data */
};
```

The semantics of the first six fields in `pci_device_id` conform to the PCI parlance discussed previously. The last field `driver_data` is private to the driver and is commonly used to co-relate configuration information if the driver supports multiple cards.

The Ethernet-Modem card has a device ID and a configuration space corresponding to each of its two functions. Because the two card functions are unconnected, you need separate PCI drivers to handle them. The *drivers/net/* directory is a good place to hold the Ethernet driver, and *drivers/serial/* is the right location to place its serial counterpart. The Ethernet driver in Listing 10.1 supports the network function and announces a set of associated IDs in its `pci_device_id` table. The serial driver in Listing 10.2 is similar, except that it's responsible for the modem function. The associated class codes and class masks are left unstated by both drivers because the vendor ID/device ID combination itself uniquely identifies their functionality.

The PCI subsystem provides macros such as `PCI_DEVICE()` and `PCI_DEVICE_CLASS()` to ease the creation of the `pci_device_id` table. `PCI_DEVICE()`, for example, creates a `pci_device_id` element from the specified vendor ID and device ID. So you may rewrite `network_device_pci_table` in Listing 10.1 as follows:

```
struct pci_device_id network_driver_pci_table[] __devinitdata = {
  {PCI_DEVICE(MY_VENDOR_ID, MY_DEVICE_ID_NET)
   .driver_data = (unsigned long)network_driver_private_data},
  {0},
};
```

The `MODULE_DEVICE_TABLE()` macro in Listing 10.1 and Listing 10.2 marks the `pci_device_id` table in the module image. This information loads the module on demand when the CardBus card is inserted. We explored this mechanism in the section "Module Autoload" in Chapter 4, "Laying the Groundwork," and used it in the context of `pcmcia_device_id` in Chapter 9, "PCMCIA and Compact Flash."

When the PCI hotplug layer senses the presence of a card with properties matching the ones announced by the

pci_device_id table of a driver, it invokes the probe() method belonging to that driver. This gives an opportunity to the driver to claim the card. Obviously, PCI drivers have to associate their pci_device_id table with their probe() method. This tie up is achieved by the pci_driver structure that drivers register with the PCI subsystem during initialization. To perform this registration, drivers call pci_register_driver().

Listing 10.1. Registering the Network Function

Code View:

```
#include <linux/pci.h>

#define MY_VENDOR_ID      0xABCD
#define MY_DEVICE_ID_NET  0xEF01

/* The set of PCI cards that this driver supports. Only a single
   entry in our case. Look at include/linux/mod_devicetable.h for
   the definition of pci_device_id */
struct pci_device_id network_driver_pci_table[] __devinitdata = {
{
  { MY_VENDOR_ID,               /* Interface chip manufacturer ID */
    MY_DEVICE_ID_NET,           /* Device ID for the network
                                   function */
    PCI_ANY_ID,                 /* Subvendor ID wild card */
    PCI_ANY_ID,                 /* Subdevice ID wild card */
    0, 0,                       /* class and classmask are
                                   unspecified */
    network_driver_private_data /* Use this to co-relate
                                   configuration information if the
                                   driver supports multiple
                                   cards. Can be an enumerated type. */
  }, {0},
};

/* struct pci_driver is defined in include/linux/pci.h */
struct pci_driver network_pci_driver = {
  .name     =  "ntwrk",                     /* Unique name */
  .probe    =  net_driver_probe,            /* See Listing 10.3 */
  .remove   = __devexit_p(net_driver_remove),/* See Listing 10.3 */
  .id_table = network_driver_pci_table,     /* See above */

  /* suspend() and resume() methods that implement power
     management are not used by this driver */

};

/* Ethernet driver initialization */
static int __init
network_driver_init(void)
{
  pci_register_driver(&network_pci_driver);
  return 0;
}

/* Ethernet driver exit */
static void __exit
network_driver_exit(void)
{
  pci_unregister_driver(&network_pci_driver);
}
```

```
module_init(network_driver_init);
module_exit(network_driver_exit);
MODULE_DEVICE_TABLE(pci, network_driver_pci_table);
```

Listing 10.2. Registering the Modem Function

```
Code View:
#include <linux/pci.h>

#define MY_VENDOR_ID      0xABCD
#define MY_DEVICE_ID_MDM  0xEF02

/* The set of PCI cards that this driver supports */
struct pci_device_id modem_driver_pci_table[] __devinitdata = {
{
  { MY_VENDOR_ID,               /* Interface chip manufacturer ID */
    MY_DEVICE_ID_MDM,           /* Device ID for the modem
                                   function */
    PCI_ANY_ID,                 /* Subvendor ID wild card */
    PCI_ANY_ID,                 /* Subdevice ID wild card */
    0, 0,                       /* class and classmask are
                                   unspecified */
    modem_driver_private_data   /* Use this to co-relate
                                   configuration information if the driver
                                   supports multiple cards. Can be an
                                   enumerated type. */
  }, {0},
};

struct pci_driver modem_pci_driver = {
  .name     =  "mdm",                          /* Unique name */
  .probe    =  modem_driver_probe,             /* See Listing 10.4 */
  .remove   = __devexit_p(modem_driver_remove),/* See Listing 10.4 */
  .id_table =  modem_driver_pci_table,         /* See above */
  /* suspend() and resume() methods that implement power
   management are not used by this driver */
};

/* Modem driver initialization */
static int __init
modem_driver_init(void)
{
  pci_register_driver(&modem_pci_driver);
  return 0;
}
/* Modem driver exit */
static void __exit
modem_driver_exit(void)
{
  pci_unregister_driver(&modem_pci_driver);
}

module_init(modem_driver_init);
module_exit(modem_driver_exit);
```

```
MODULE_DEVICE_TABLE(pci, modem_driver_pci_table);
```

Listing 10.3 implements the `probe()` method for the network function. This

- Enables the PCI device
- Discovers resource information such as I/O base addresses and IRQ
- Allocates and populates a networking data structure associated with this device
- Registers itself with the kernel networking layer

Listing 10.4 performs similar work for the modem function. In this case, the driver registers with the kernel serial layer instead of the networking layer.

Listings 10.3 and 10.4 also implement `remove()` methods, which are invoked when the CardBus card is ejected or when the driver module is unloaded. `remove()` is the reverse of `probe()`; it frees resources and unregisters the driver from relevant subsystems. The `__devexit_p()` macro that Listing 10.1 uses to declare the `remove()` method discards the supplied function at compile time if you haven't enabled hotplug support and if the driver is not a dynamically loadable module.

The PCI subsystem calls `probe()` with two arguments:

1. A pointer to `pci_dev`, the data structure that describes this PCI device. This structure, defined in *include/linux/pci.h*, is maintained by the PCI subsystem for each PCI device on your system.
2. A pointer to `pci_device_id`, the entry in the driver's `pci_device_id` table that matches the information found in the configuration space of the inserted card.

Listing 10.3. Probing the Network Function

Code View:

```
#include <linux/pci.h>

unsigned long ioaddr;

/* Probe method */
static int __devinit
net_driver_probe(struct pci_dev *pdev,
                 const struct pci_device_id *id)
{
  /* The net_device structure is defined in include/linux/netdevice.h.
     See Chapter 15, "Network Interface Cards", for the description */
```

```
    struct net_device *net_dev;

  /* Ask low-level PCI code to enable I/O and memory regions for
     this device. Look up the IRQ for the device that the PCI
     subsystem allotted when it walked the bus */
  pci_enable_device(pdev);

  /* Use this device in bus mastering mode, since the network
     function of this card is capable of DMA */
  pci_set_master(pdev);

  /* Allocate an Ethernet interface and fill in generic values in
     the net_dev structure. prv_data is the private driver data
     structure that contains buffers, locks, and so on. This is
     left undefined. Wait until Chapter 15 for more on
     alloc_etherdev() */
  net_dev = alloc_etherdev(sizeof(struct prv_data));

  /* Populate net_dev with your network device driver methods */
  net_dev->hard_start_xmit = &mydevice_xmit; /* See Listing 10.6 */

  /* More net_dev initializations */
  /* ... */

  /* Get the I/O address for this PCI region. All card registers
     specified in Table 10.3 are assumed to be in bar 0 */
  ioaddr = pci_resource_start(pdev, 0);

  /* Claim a 128-byte I/O region */
  request_region(ioaddr, 128, "ntwrk");
  /* Fill in resource information obtained from the PCI layer */
  net_dev->base_addr = ioaddr;
  net_dev->irq       = pdev->irq;
  /* ... */
  /* Setup DMA. Defined in Listing 10.5 */
  dma_descriptor_setup(pdev);

  /* Register the driver with the network layer. This will allot
     an unused ethX interface */
  register_netdev(net_dev);

  /* ... */
}

/* Remove method */
static void __devexit
net_driver_remove(struct pci_dev *pdev)
{
  /* Free transmit and receive DMA buffers.
     Defined in Listing 10.5 */
  dma_descriptor_release(pdev);

  /* Release memory regions */
  /* ... */

  /* Unregister from the networking layer */
  unregister_netdev(dev);
  free_netdev(dev);
```

```
  /* ... */
}
```

Listing 10.4. Probing the Modem Function

```
Code View:
/* Probe method */
static int __devinit
modem_driver_probe(struct pci_dev *pdev,
                   const struct pci_device_id *id)
{
  struct uart_port port; /* See Chapter 6, "Serial Drivers" */
  /* Ask low-level PCI code to enable I/O and memory regions
     for this PCI device */
  pci_enable_device(pdev);

  /* Get the PCI IRQ and I/O address to be used and populate the
     uart_port structure (see Chapter 6) with these resources. Look at
     include/linux/pci.h for helper functions */

  /* ... */

  /* Register this information with the serial layer and get an
     unused ttySX port allotted to the card. Look at Chapter 6 for
     more on serial drivers */
  serial8250_register_port(&port);

  /* ... */
}

/* Remove method */
static void __devexit
modem_driver_remove(struct pci_dev *dev)
{
  /* Unregister the port from the serial layer */
  serial8250_unregister_port(&port);

  /* Disable device */
  pci_disable_device(dev);
}
```

To recap, let's trace the code path from the time you insert the Ethernet-Modem CardBus card until you are allotted a network interface (ethX) and a serial port (*/dev/ttySX*):

1. For each supported CardBus function, the corresponding driver initialization routine registers a `pci_device_id` table of supported cards and a `probe()` routine. This is shown in Listing 10.1 and Listing 10.2.

2. The PCI hotplug layer detects card insertion and gleans the vendor ID and device ID of each device function from the card's PCI configuration space.

3. Because the IDs match with those registered by the card's Ethernet and serial drivers, the corresponding `probe()` methods are invoked. This results in the invocation of `net_driver_probe()` and `modem_driver_probe()` described respectively in Listing 10.3 and Listing 10.4.

4. The `probe()` methods configure the Ethernet and modem portions of the PCI driver with resource information. This leads to the allocation of a networking interface (ethX) and a serial port (ttySX) to the card. From this point on, application data flows over these interfaces.

Data Transfer

The network function belonging to the sample CardBus device uses the following strategy for data transfer: The card expects the device driver to supply it with an array of two receive DMA descriptors and an array of two transmit DMA descriptors. Each DMA descriptor contains the address of an associated data buffer, its length, and a control word. You can use the control word to tell the device whether the descriptor contains valid data. For a transmit descriptor, you may also program it to request an interrupt after data transmission. The card looks for a valid descriptor and DMA's data to/from the associated data buffer. To suit this elementary scheme, the example driver uses only the coherent DMA interface. The driver coherently allocates a large buffer that holds the descriptors and their associated data buffers. The receive and transmit buffers are 1536 bytes long to match the *maximum transmission unit* (MTU) of Ethernet frames. The descriptors and buffers are pictorially shown in Figure 10.2. The top 24 bytes of each array in the figure hold two 12-byte DMA descriptors, and the rest of the memory is occupied by two 1536-byte DMA buffers. The 12-byte descriptor layout shown in the figure is assumed to match the format specified in the card's data sheet.

Figure 10.2. DMA descriptors and buffers for the CardBus device.

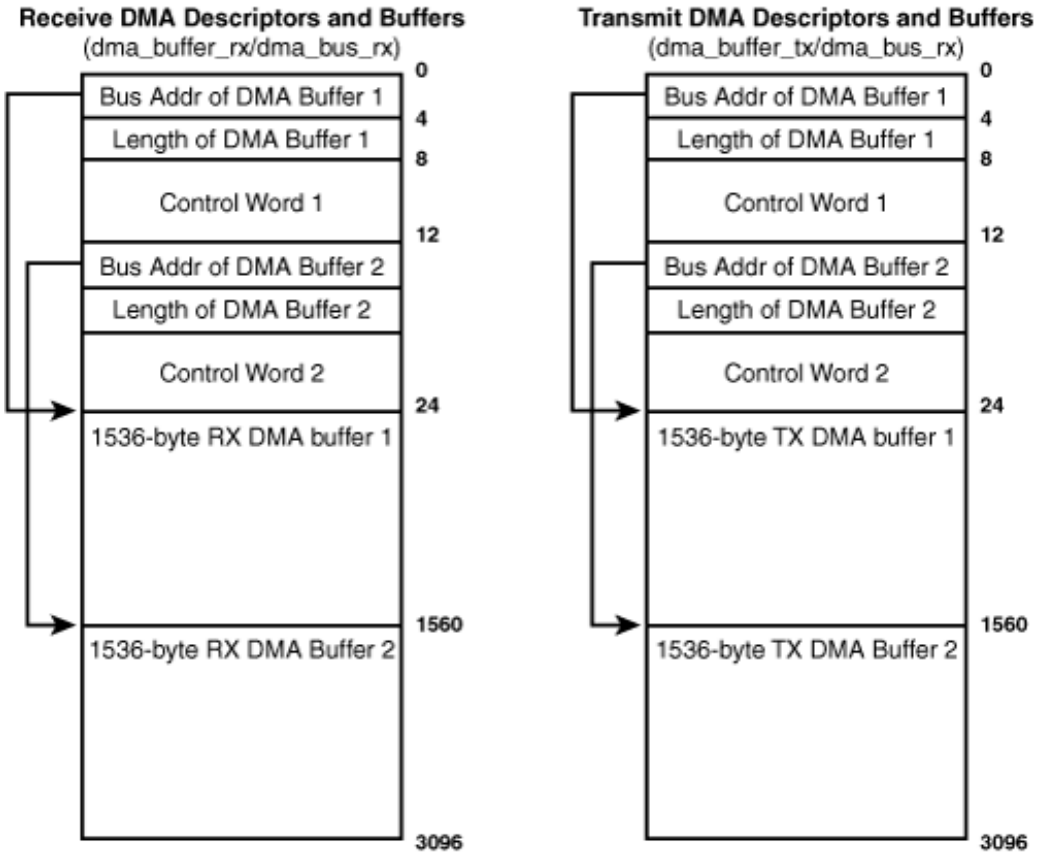


Table 10.3 shows the register layout of the card's network function.

Table 10.3. Register Layout of the Card's Network Function

Register Name	Description	Offset into I/O Space
`DMA_RX_REGISTER`	Holds the bus address of the receive DMA descriptor array (`dma_bus_rx`)	`0x0`
`DMA_TX_REGISTER`	Holds the bus address of the transmit DMA descriptor array (`dma_bus_tx`)	`0x4`
`CONTROL_REGISTER`	Control word that commands the card to initiate DMA, stop DMA, and so on	`0x8`

Listing 10.5. Setting Up DMA Descriptors and Buffers

Code View:

```
/* Device-specific data structure for the Ethernet Function */
struct device_data {
  struct pci_dev *pdev;     /* The PCI Device structure */
  struct net_device *ndev;  /* The Net Device structure */
  void *dma_buffer_rx;      /* Kernel virtual address of the
                               receive descriptor */
```

```
  dma_addr_t dma_bus_rx;    /* Bus address of the receive
                               descriptor */
  void *dma_buffer_tx;      /* Kernel virtual address of the
                               transmit descriptor */
  dma_addr_t dma_bus_tx;    /* Bus address of the transmit
                               descriptor */
  /* ... */
  spin_lock_t device_lock;  /* Serialize */
} *mydev_data;
/* On-card registers related to DMA */
#define DMA_RX_REGISTER_OFFSET  0x0 /* Offset of the register
                                       holding the bus address
                                       of the RX descriptor */
#define DMA_TX_REGISTER_OFFSET  0x4 /* Offset of the register
                                       holding the bus address
                                       of the TX descriptor */
#define CONTROL_REGISTER        0x8 /* Offset of the control
                                       register */

/* Control Register Defines */
#define INITIATE_XMIT           0x1

/* Descriptor control word definitions */
#define FREE_FLAG               0x1 /* Free Descriptor */
#define INTERRUPT_FLAG          0x2 /* Assert interrupt after DMA */

/* Invoked from Listing 10.3 */
static void
dma_descriptor_setup(struct pci_dev *pdev)
{
  /* Allocate receive DMA descriptors and buffers */
  mydev_data->dma_buffer_rx =
     pci_alloc_consistent(pdev, 3096, &mydev_data->dma_bus_rx);

  /* Fill the two receive descriptors as shown in Figure 10.2 */
  /* RX descriptor 1 */
  mydev_data->dma_buffer_rx[0] = cpu_to_le32((unsigned long)
              (mydev_data->dma_bus_rx + 24));  /* Buffer address */
  mydev_data->dma_buffer_rx[1] = 1536;         /* Buffer length */
  mydev_data->dma_buffer_rx[2] = FREE_FLAG;    /* Descriptor is free */

  /* RX descriptor 2 */
  mydev_data->dma_buffer_rx[3] = cpu_to_le32((unsigned long)
            (mydev_data->dma_bus_rx + 1560)); /* Buffer address */
  mydev_data->dma_buffer_rx[4] = 1536;         /* Buffer length */
  mydev_data->dma_buffer_rx[5] = FREE_FLAG;    /* Descriptor is free */

  wmb(); /* Write Memory Barrier */
  /* Write the address of the receive descriptor to the appropriate
     register in the card. The I/O base address, ioaddr, was populated
     in Listing 10.3 */
 outl(cpu_to_le32((unsigned long)mydev_data->dma_bus_rx),
                  ioaddr + DMA_RX_REGISTER_OFFSET);
  /* Allocate transmit DMA descriptors and buffers */
  mydev_data->dma_buffer_tx =
      pci_alloc_consistent(pdev, 3096, &mydev_data->dma_bus_tx);

  /* Fill the two transmit descriptors as shown in Figure 10.2 */
  /* TX descriptor 1 */
```

```
  mydev_data->dma_buffer_tx[0] = cpu_to_le32((unsigned long)
              (mydev_data->dma_bus_tx + 24));   /* Buffer address */
  mydev_data->dma_buffer_tx[1] = 1536;          /* Buffer length */
  /* Valid descriptor. Generate an interrupt
     after completing the DMA */
  mydev_data->dma_buffer_tx[2] = (FREE_FLAG | INTERRUPT_FLAG);
  /* TX descriptor 2 */
  mydev_data->dma_buffer_tx[3] = cpu_to_le32((unsigned long)
              (mydev_data->dma_bus_tx + 1560)); /* Buffer address */
  mydev_data->dma_buffer_tx[4] = 1536;          /* Buffer length */
  mydev_data->dma_buffer_tx[5] = (FREE_FLAG | INTERRUPT_FLAG);

  wmb(); /* Write Memory Barrier */
  /* Write the address of the transmit descriptor to the appropriate
     register in the card. The I/O base, ioaddr, was populated in
     Listing 10.3 */
  outl(cpu_to_le32((unsigned long)mydev_data->dma_bus_tx),
                  ioaddr + DMA_TX_REGISTER_OFFSET);
}

/* Invoked from Listing 10.3 */
static void
dma_descriptor_release(struct pci_dev *pdev)
{
  pci_free_consistent(pdev, 3096, mydev_data->dma_bus_tx);
  pci_free_consistent(pdev, 3096, mydev_data->dma_bus_rx);
}
```

Listing 10.5 enforces a write barrier by calling `wmb()` to prevent the CPU from reordering the `outl()` before populating the DMA descriptor. On an x86 processor, `wmb()` reduces to a NOP because Intel CPUs enforce writes in program order. When writing the DMA descriptor address to the card and when populating the buffer's bus address inside the DMA descriptor, the driver converts the native byte order to PCI little-endian format using `cpu_to_le32()`. On Intel CPUs, this again has no effect because both PCI and Intel processors communicate in little-endian. On several other architectures, for example, an ARM9 CPU running in the big-endian mode, both `wmb()` and `cpu_to_le32()` assume significance.

Now that you have the descriptors and buffers mapped and ready to go, it's time to look at how data is exchanged between the system and the CardBus device, as shown in Listing 10.6. We won't dwell on the network interfaces and networking data structures because Chapter 15 is devoted to doing that.

Listing 10.6. Receiving and Transmitting Data

Code View:

```
/* The interrupt handler */
static irqreturn_t
mydevice_interrupt(int irq, void *devid)
{
  struct sk_buff *skb;
  /* ... */
  /* If this is a receive interrupt, collect the packet and pass it
     on to higher layers. Look at the control word in each RX DMA
     descriptor to figure out whether it contains data. Assume for
     convenience that the first RX descriptor was used by the card
```

```
     to DMA this received packet */

  packet_size = mydev_data->dma_buffer_rx[1];
  /* In real world drivers, the sk_buff is pre-allocated, stream-
     mapped, and supplied to the card after setting the FREE_FLAG
     during device open(). The received data is directly
     DMA'ed to this sk_buff instead of the memcpy() performed here,
     as you will learn in Chapter 15. The card clears the FREE_FLAG
     before the DMA */
  skb = dev_alloc_skb(packet_size); /* See Figure 15.2 of Chapter 15 */
  skb->dev = mydev_data->ndev;      /* Owner network device */
  memcpy(skb, mydev_data->dma_buffer_rx[24], packet_size);
  /* Pass the received data to higher-layer protocols */
  skb_put(skb, packet_size);
  netif_rx(skb);
  /* ... */
  /* Make the descriptor available to the card again */
  mydev_data->dma_buffer_rx[2] |= FREE_FLAG;
}

/* This function is registered in Listing 10.3 and is called from
   the networking layer. More on network device interfaces in
   Chapter 15 */
static int
mydevice_xmit(struct sk_buff *skb, struct net_device *dev)
{
  /* Use a valid TX descriptor. Look at Figure 10.2 */
  /* Locking has been omitted for simplicity */
  if (mydev_data->dma_buffer_tx[2] & FREE_FLAG) {
    /* Use first TX descriptor */
    /* In the real world, DMA occurs directly from the sk_buff as
       you will learn later on! */
    memcpy(mydev_data->dma_buffer_tx[24], skb->data, skb->len);
    mydev_data->dma_buffer_tx[1] = skb->len;
    mydev_data->dma_buffer_tx[2] &= ~FREE_FLAG;
    mydev_data->dma_buffer_tx[2] |= INTERRUPT_FLAG;
  } else if (mydev_data->dma_buffer[5] & FREE_FLAG) {
    /* Use second TX descriptor */
    memcpy(mydev_data->dma_buffer_tx[1560], skb->data, skb->len);
    mydev_data->dma_buffer_tx[4] = skb->len;
    mydev_data->dma_buffer_tx[5] &= ~FREE_FLAG;
    mydev_data->dma_buffer_tx[5] |= INTERRUPT_FLAG;
  } else {
    return –EIO; /* Signal error to the caller */
  }
  wmb(); /* Don't reorder writes across this barrier */

  /* Ask the card to initiate DMA. ioaddr is defined
     in Listing 10.3 */
  outl(INITIATE_XMIT, ioaddr + CONTROL_REGISTER);
}
```

When the CardBus device receives an Ethernet packet, it DMAs it to a free RX descriptor and interrupts the CPU. The interrupt handler `mydevice_interrupt()` collects the packet from the receive DMA buffer, copies it to a networking data structure (`sk_buff`), and passes it on to higher protocol layers.

The transmit routine `my_device_xmit()` is responsible for initiating DMA in the reverse direction. It DMAs transmit packets to card memory. For this, `my_device_xmit()` chooses a TX DMA descriptor that is unused by the card (or whose `FREE_FLAG` is set) and uses the associated transmit buffer for data transfer. `FREE_FLAG` is cleared soon after, signaling that the descriptor now belongs to the card. The descriptor is released in the interrupt handler (`FREE_FLAG` is set again) when the card asserts an interrupt after completing the transmit (not shown in Listing 10.6).

This example driver uses a simplified buffer management scheme that is not performance-sensitive. High-speed network drivers implement a more elaborate plan that employs a combination of coherent and streaming DMA mappings. They maintain linked lists of transmit and receive descriptors and implement free and in-use pools for buffer management. Their receive and transmit data structures look like this:

Code View:
```
/* Ring of receive buffers */
struct rx_list {
  void *dma_buffer_rx;        /* Kernel virtual address of the
                                 transmit descriptor */
  dma_addr_t dma_bus_rx;      /* Bus address of the transmit
                                 descriptor */
  unsigned int size;          /* Buffer size */
  struct list_head next_desc; /* Pointer to the next element */
  struct sk_buff *skb;        /* Network Packet */
  dma_addr_t sk_bus;          /* Bus address of network packet */
} *rxlist;

/* Ring of transmit buffers */
struct tx_list {
  void *dma_buffer_tx;        /* Kernel virtual address of the
                                 receive descriptor */
  dma_addr_t dma_bus_tx;      /* Bus address of the transmit
                                 descriptor */
  unsigned int size;          /* Buffer size */
  struct list_head next_desc; /* Pointer to the next element */
  struct sk_buff *skb;        /* Network Packet */
  dma_addr_t sk_bus;          /* Bus address of network packet */
} *txlist;
```

The receive and transmit DMA descriptors (`rxlist->dma_buffer_rx` and `txlist->dma_buffer_tx`) are mapped coherently as done in Listing 10.5. The payload buffers (`rxlist->skb->data` and `txlist->skb->data`) are, however, mapped using streaming DMA. The receive buffers are preallocated and stream mapped into a free pool during device open, while the transmit buffers are mapped on-the-fly. This avoids the extra data copy performed by `mydevice_interrupt()` from the coherently mapped receive DMA buffer to the network buffer (and the extra copy done by `mydevice_xmit()` in the reverse direction).

Code View:
```
/* Preallocating/replenishing receive buffers. Also see the section, "Buffer
   Management and Concurrency Control" in Chapter 15 */
/* ... */
struct sk_buff *skb = dev_alloc_skb();
skb_reserve(skb, NET_IP_ALIGN);
/* Map using streaming DMA */
rxlist->sk_bus = pci_map_single(pdev, rxlist->skb->data,
                                rxlist->skb->len, PCI_DMA_TODEVICE);
```

```
/* Allocate a DMA descriptor and populate it with the address mapped
   above. Add the descriptor to the receive descriptor ring */
/* ... */
```

Debugging

Enable *Bus Options* → *PCI Support* → *PCI Debugging* in the kernel configuration menu to ask the PCI core to emit debug messages. Explore */proc/bus/pci/devices* and */sys/devices/pciX:Y/* for information about PCI devices on your system such as the CardBus Ethernet-Modem card discussed in this chapter. */proc/interrupts* lists IRQs active on your system, including those used by the PCI layer.

As you saw, lspci gleans information about all PCI buses and devices on your system. You may also use it to dump the configuration space of PCI cards.

A PCI bus analyzer can help debug low-level problems and tune performance.

Looking at the Sources

PCI core and bus access routines live in *drivers/pci/.* To obtain a list of helper routines offered by the PCI subsystem, search for `EXPORT_SYMBOL` inside this directory. For definitions and prototypes related to the PCI layer, look at *include/linux/pci*.h*.

You can spot several PCI device drivers in subdirectories under *drivers/net/*, *drivers/scsi/*, and *drivers/video/.* To locate all PCI drivers, recursively grep the *drivers/* tree for `pci_register_driver()`.

If you do not find a good starting point to develop a custom PCI network driver, begin with the skeletal PCI network driver *drivers/net/pci-skeleton.c.* For a brief tutorial on PCI programming, look at *Documentation/pci.txt.* For a description of the PCI DMA API, read *Documentation/DMA-mapping.txt*.

Table 10.4 summarizes the main data structures used in this chapter. Table 10.5 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 10.4. Summary of Data Structures

Data Structure	Location	Description
`pci_dev`	*include/linux/pci.h*	Representation of a PCI device
`pci_driver`	*include/linux/pci.h*	Representation of a PCI driver
`pci_device_id`	*include/linux/mod_devicetable.h*	Identity of a PCI card
`dma_addr_t`	*include/asm-your-arch/types.h*	Bus address of a DMA buffer
`scatterlist`	*include/asm-your-arch/scatterlist.h*	Scatter-gather list of DMA buffers
`sk_buff`	*include/linux/skbuff.h*	Main networking data structure (see Chapter 15 for more explanations)

Table 10.5. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description

`pci_read_config_byte()`
`pci_read_config_word()`
`pci_read_config_dword()`
`pci_write_config_byte()`
`pci_write_config_word()`
`pci_write_config_dword()`
	include/linux/pci.h
	drivers/pci/access.c
		Routines to operate on the PCI configuration space.

`pci_resource_start()`
`pci_resource_len()`
`pci_resource_end()`
`pci_resource_flags()`
	include/linux/pci.h	These routines operate on PCI I/O and memory regions to obtain the base address, length, end address, and control flags.

`pci_request_region()`	*drivers/pci/pci.c*	Reserves PCI I/O or memory regions.

Kernel Interface	Location	Description

`ioremap()`
`ioremap_nocache()`
`pci_iomap()`
	include/asm-your-arch/io.h
	
	arch/your-arch/mm/ioremap.c
	
	lib/iomap.c
		Obtains CPU access to device memory.

`pci_set_dma_mask()`	*drivers/pci/pci.c*	If this function returns success, you may DMA to any address within the mask specified as argument.
`pci_alloc_consistent()`	*include/asm-generic/pci-dma-compat.h include/asm-your-arch/dma-mapping.h*	Obtains a cache-coherent DMA buffermapping.
`pci_free_consistent()`	*include/asm-generic/pci-dma-compat.h include/asm-your-arch/dma-mapping.h*	Unmaps a cache-coherent DMA buffer.
`pci_map_single()`	*include/asm-generic/pci-dma-compat.h include/asm-your-arch/dma-mapping.h*	Obtains a streaming DMA buffer mapping.
`pci_unmap_single()`	*include/asm-generic/pci-dma-compat.h include/asm-your-arch/dma-mapping.h*	Unmaps a streaming DMA buffer.
`pci_dma_sync_single()`	*include/asm-generic/pci-dma-compat.h include/asm-your-arch/dma-mapping.h*	Synchronizes a streaming DMA buffer so that the CPU can reliably operate on it.

`pci_map_sg()`
`pci_unmap_sg()`
`pci_dma_sync_sg()`
	include/asm-generic/pci-dma-compat.h include/asm-your-arch/dma-mapping.h	Maps/unmaps/synchronizes a scatter-gather list of streaming DMA buffers.

`pci_register_driver()`	*include/linux/pci.h* *drivers/pci/pci-driver.c*	Registers a driver with the PCI core.
`pci_unregister_driver()`	*drivers/pci/pci-driver.c*	Unregisters a driver from the PCI core.
`pci_enable_device()`	*drivers/pci/pci.c*	Asks low-level PCI code to enable I/O and memory regions for this device.

Kernel Interface	Location	Description
`pci_disable_device()`	*drivers/pci/pci.c*	Reverse of `pci_enable_device()`.
`pci_set_master()`	*drivers/pci/pci.c*	Sets the device in DMA bus-mastering mode.

Chapter 11. Universal Serial Bus

In This Chapter

- USB Architecture 312
- Linux-USB Subsystem 317
- Driver Data Structures 317
- Enumeration 324
- Device Example: Telemetry Card 324
- Class Drivers 338
- Gadget Drivers 348
- Debugging 349
- Looking at the Sources 351

U*niversal serial bus* (USB) is the de facto external bus in today's computers. USB, with its support for hotplugging, generic class drivers, and versatile data-transfer modes, is the usual route in the consumer electronics space to bring a diverse spectrum of technologies to computer systems. Its sweeping popularity and the accompanying economics of volume have played a part in fueling the adoption and acceptance of computer peripheral technologies around the world.

USB Architecture

USB is a master-slave protocol where a host controller communicates with client devices. Figure 11.1 shows USB in the PC environment. The USB host controller is part of the South Bridge chipset and communicates with the processor over the PCI bus.

Figure 11.1. USB in the PC environment.

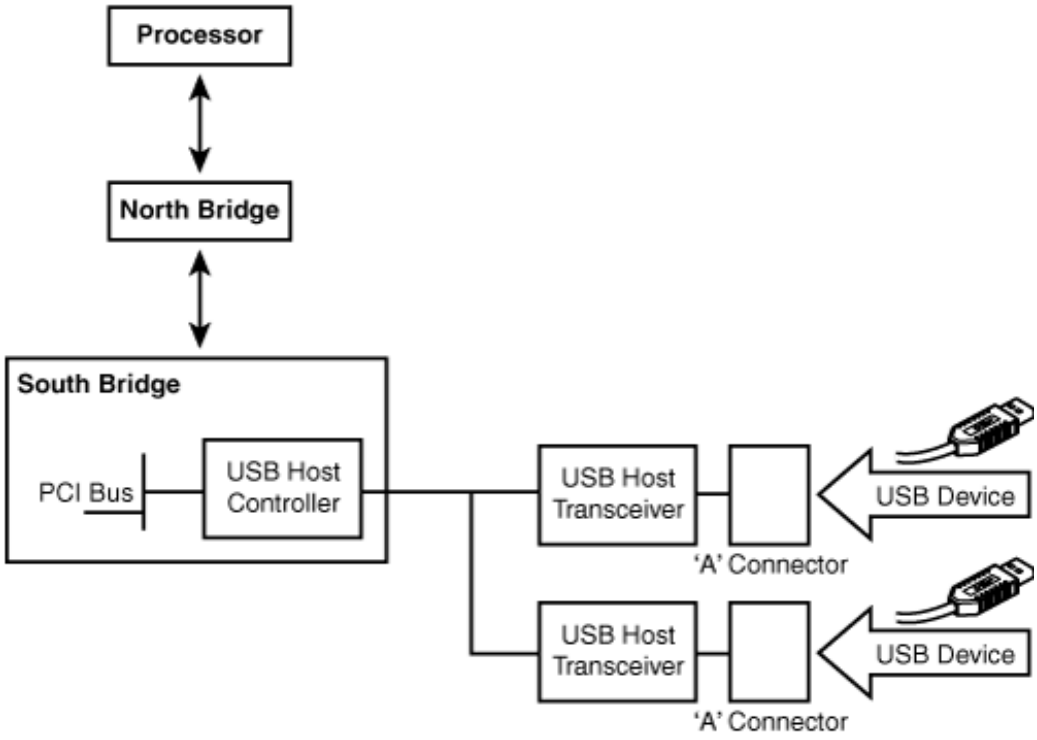


Figure 11.2 illustrates USB on an embedded device. The SoC in the figure has built-in USB controller silicon that supports four buses and three modes of operation:

- Bus 1 runs in host mode and is wired to an A-type receptacle via a USB transceiver (see the sidebar "USB Receptacles and Transceivers"). You can connect a USB pen drive or a keyboard to this port.

- Bus 2 also functions in host mode but the associated transceiver is connected to an internal USB device rather than to a receptacle. Examples of internal USB devices are biometric scanners, cryptographic engines, printers, *Disk-On-Chips* (DOCs), touch controllers, and telemetry cards.

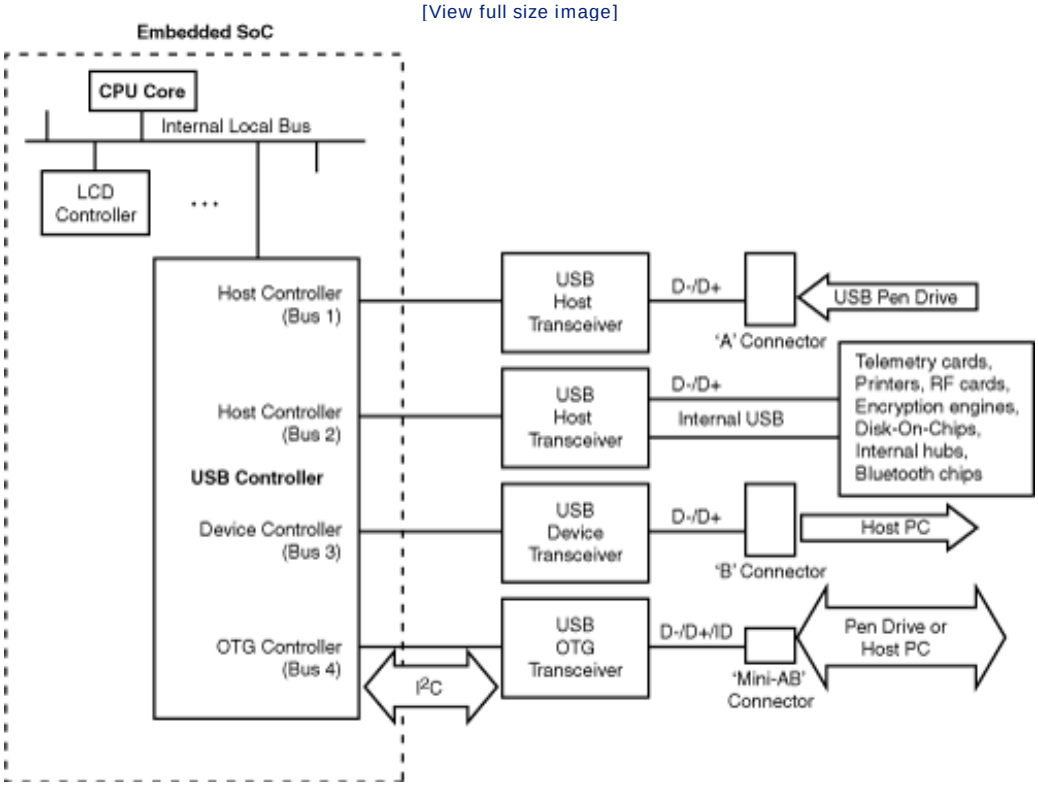
- Bus 3 runs in device mode and is wired to a B-type receptacle through a transceiver. The B-type receptacle connects to a host computer via a B-to-A cable. In this mode, the embedded device functions as, for example, a USB pen drive, and exports a storage partition to the outside world. Embedded devices such as MP3 players and cell phones are more likely than PC systems to be at the device side of USB, so many embedded SoCs support a USB device controller in addition to a host controller.

- Bus 4 is driven by an *On-The-Go* (OTG) controller. You can use this port, for example, to either connect a pen drive to your system or to turn your system into a pen drive and connect it to a host. Unlike buses 1

to 3, bus 4 uses an intelligent transceiver that exchanges control information with the processor over I^2C. The transceiver is wired to a Mini-AB OTG receptacle. If two embedded devices support OTG, they can directly communicate without the intervention of a host computer.

Figure 11.2. USB on an embedded system.

[View full size image]



Most of this chapter is written from the perspective of a system residing at the host-side of USB. We briefly look at the device function in the section "Gadget Drivers." Mainstream *host controller drivers* (HCDs) are already available, so in this chapter we further confine ourselves to drivers for USB devices (also called *client drivers*).

USB Receptacles and Transceivers

USB hosts use four-pin A-type rectangular receptacles, whereas USB devices connect via four-pin B-type square receptacles. In both cases, the four pins are differential data signals D+ and D-, a voltage line VBUS, and ground. VBUS is used to supply power from USB hosts to USB devices. VBUS is thus pulled high on an A connector but receives power on a B connector. USB OTG controllers connect to five-pin Mini-AB rectangular receptacles having a smaller form factor. Four of the Mini-AB pins are identical to what we discussed previously; the fifth is an ID pin used to detect whether the connected peripheral is a host or a device.

The same transceiver chip (such as TUSB1105 from Texas Instruments) can be used on USB hosts and devices. You may thus choose to use the same transceiver part on buses 1 through 3 in Figure 11.2. OTG requires a special-purpose transceiver chip (such as ISP1301 from Philips Semiconductors), however.

Bus Speeds

USB supports three operational speeds. The original USB 1.0 specification supports 1.5MBps, referred to as low-speed USB. USB 1.1, the next version of the specification, handles 12MBps, called full-speed USB. The current level of the specification is USB 2.0, which supports 480MBps, or high-speed USB. USB 2.0 is backward-compatible with the earlier versions of the specification. Peripherals such as USB keyboards and mice are examples of low-speed devices, and USB storage drives are examples of full-speed and high-speed devices. Today's PC systems are USB 2.0-compliant and allow all three target speeds, but some embedded controllers adhere to USB 1.1 and support only full-speed and low-speed modes of operation.

Host Controllers

USB host controllers conform to one of a few standards:

- Universal Host Controller Interface (UHCI): The UHCI specification was initiated by Intel, so your PC is likely to have this controller if it's Intel-based.
- Open Host Controller Interface (OHCI): The OHCI specification originated from companies such as Compaq and Microsoft. An OHCI-compatible controller has more intelligence built in to hardware than UHCI, so an OHCI HCD is relatively simpler than a UHCI HCD.
- Enhanced Host Controller Interface (EHCI): This is the host controller that supports high-speed USB 2.0 devices. EHCI controllers usually have either a UHCI or OHCI companion controller to handle slower devices.
- USB OTG controllers: They are getting increasingly popular in embedded microcontrollers. With OTG support, each communicating end can act as a *dual-role device* (DRD). By initiating a dialog using the *Host Negotiation Protocol* (HNP), a DRD can switch itself to host mode or device mode based on the desired functionality.

In addition to these mainstream USB host controllers, Linux supports a few more controllers. An example is the HCD for the ISP116x chip.

Host controllers have a built-in hardware component called the *root hub.* The root hub is a virtual hub that sources USB ports. The ports, in turn, can connect to external or internal physical hubs and source more ports,

yielding a tree topology.

Transfer Types

Data exchange with a USB device can be one of four types:

- Control transfers, used to carry configuration and control information
- Bulk transfers that ferry large quantities of time-insensitive data
- Interrupt transfers that exchange small quantities of time-sensitive data
- Isochronous transfers for real-time data at predictable bit rates

A USB storage drive, for example, uses control transfers to issue disk access commands and bulk transfers to exchange data. A keyboard uses interrupt transfers to carry key strokes within predictable delays. A device that needs to stream audio data in real time uses isochronous transfers. The responsibilities of the four transfer types for USB Bluetooth devices are discussed in the section "Device Example: USB Adapter" in Chapter 16, "Linux Without Wires."

Addressing

Each addressable unit in a USB device is called an *endpoint.* The address assigned to an endpoint is called an *endpoint address.* Each endpoint address has an associated data transfer type. If an endpoint is responsible for bulk data transfer, for example, it's called a *bulk endpoint.* Endpoint address 0 is used exclusively for device configuration. A control pipe is attached to this endpoint for device enumeration (see the section "Enumeration").

An endpoint can be associated with upstream or downstream data transfer. Data arriving upstream from a device is called an `IN` transfer, whereas data flowing downstream to a device is an `OUT` transfer. `IN` and `OUT` transfers own separate address spaces. So, you can have a bulk `IN` endpoint and a bulk `OUT` endpoint answering to the same address.

USB resembles I^2C on some counts and PCI on others as summarized in Table 11.1. USB's device addressing is similar to I^2C, while it supports hotplugging like PCI. USB device addresses, like standard I^2C, do not consume a portion of the CPU's address space. Rather, they reside in a private space ranging from 1 to 127.

Table 11.1. USB's Similarities with I^2C and PCI

USB's similarities with I^2C:

- USB and I^2C are master-slave protocols.
- Device addresses reside in a private 7-bit space.
- Device-resident memory is not mapped to the CPU's memory or I/O space, so it does not consume CPU resources.

USB's similarities with PCI:

- Devices can be hotplugged.
- Device driver architecture resembles PCI drivers. Both classes of drivers own `probe()/disconnect()`[1] methods and ID tables identifying the devices they support.
- Supports high speeds. Slower than PCI, however. See Table 10.1 in Chapter 10, "Peripheral Component Interconnect," for the speeds supported by different members of the PCI family.
- USB host controllers, like PCI controllers, usually have built-in DMA engines that can master the bus.
- Supports multifunction devices. USB supports interface descriptors per function. Each PCI device function has its own device ID and configuration space.

[1] `disconnect()` is called `remove()` in PCI parlance.

Chapter 11. Universal Serial Bus

In This Chapter

- USB Architecture 312
- Linux-USB Subsystem 317
- Driver Data Structures 317
- Enumeration 324
- Device Example: Telemetry Card 324
- Class Drivers 338
- Gadget Drivers 348
- Debugging 349
- Looking at the Sources 351

U*niversal serial bus* (USB) is the de facto external bus in today's computers. USB, with its support for hotplugging, generic class drivers, and versatile data-transfer modes, is the usual route in the consumer electronics space to bring a diverse spectrum of technologies to computer systems. Its sweeping popularity and the accompanying economics of volume have played a part in fueling the adoption and acceptance of computer peripheral technologies around the world.

USB Architecture

USB is a master-slave protocol where a host controller communicates with client devices. Figure 11.1 shows USB in the PC environment. The USB host controller is part of the South Bridge chipset and communicates with the processor over the PCI bus.

Figure 11.1. USB in the PC environment.

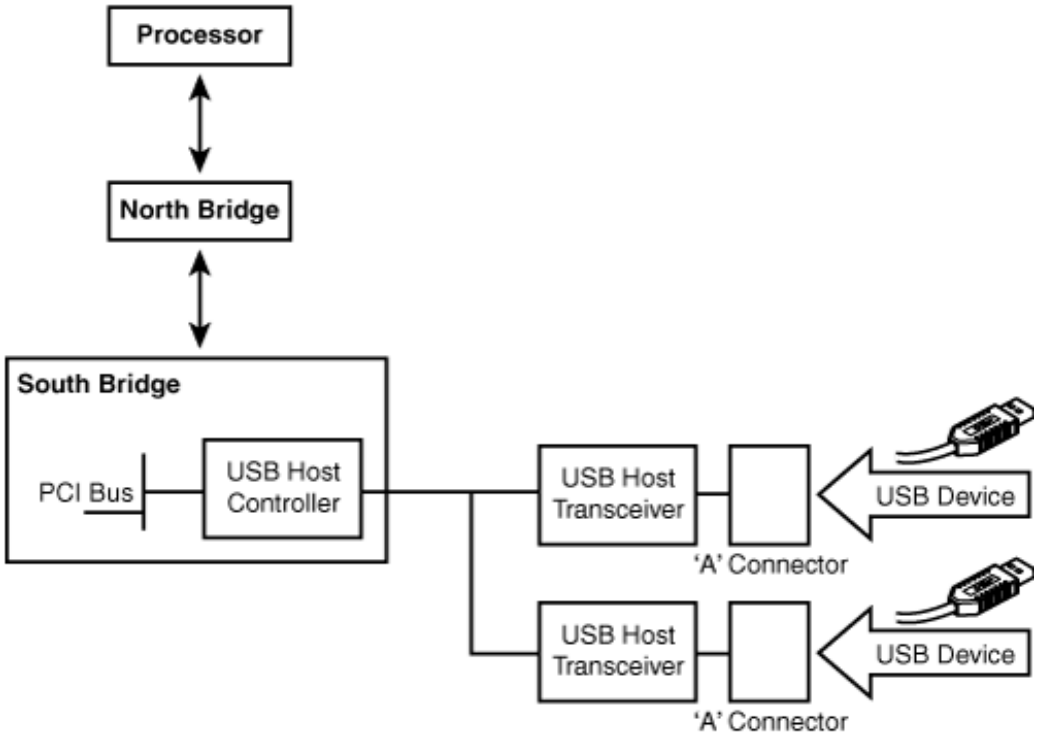


Figure 11.2 illustrates USB on an embedded device. The SoC in the figure has built-in USB controller silicon that supports four buses and three modes of operation:

- Bus 1 runs in host mode and is wired to an A-type receptacle via a USB transceiver (see the sidebar "USB Receptacles and Transceivers"). You can connect a USB pen drive or a keyboard to this port.

- Bus 2 also functions in host mode but the associated transceiver is connected to an internal USB device rather than to a receptacle. Examples of internal USB devices are biometric scanners, cryptographic engines, printers, *Disk-On-Chips* (DOCs), touch controllers, and telemetry cards.

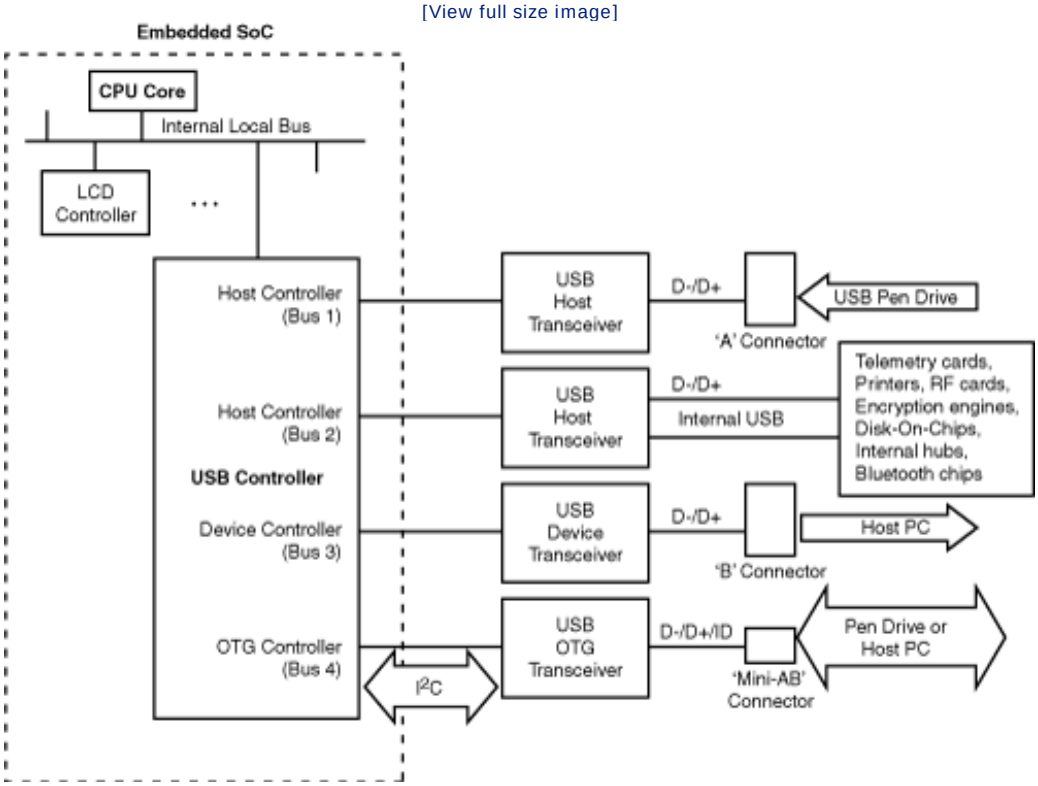
- Bus 3 runs in device mode and is wired to a B-type receptacle through a transceiver. The B-type receptacle connects to a host computer via a B-to-A cable. In this mode, the embedded device functions as, for example, a USB pen drive, and exports a storage partition to the outside world. Embedded devices such as MP3 players and cell phones are more likely than PC systems to be at the device side of USB, so many embedded SoCs support a USB device controller in addition to a host controller.

- Bus 4 is driven by an *On-The-Go* (OTG) controller. You can use this port, for example, to either connect a pen drive to your system or to turn your system into a pen drive and connect it to a host. Unlike buses 1

to 3, bus 4 uses an intelligent transceiver that exchanges control information with the processor over I^2C. The transceiver is wired to a Mini-AB OTG receptacle. If two embedded devices support OTG, they can directly communicate without the intervention of a host computer.

Figure 11.2. USB on an embedded system.

[View full size image]



Most of this chapter is written from the perspective of a system residing at the host-side of USB. We briefly look at the device function in the section "Gadget Drivers." Mainstream *host controller drivers* (HCDs) are already available, so in this chapter we further confine ourselves to drivers for USB devices (also called *client drivers*).

USB Receptacles and Transceivers

USB hosts use four-pin A-type rectangular receptacles, whereas USB devices connect via four-pin B-type square receptacles. In both cases, the four pins are differential data signals D+ and D-, a voltage line VBUS, and ground. VBUS is used to supply power from USB hosts to USB devices. VBUS is thus pulled high on an A connector but receives power on a B connector. USB OTG controllers connect to five-pin Mini-AB rectangular receptacles having a smaller form factor. Four of the Mini-AB pins are identical to what we discussed previously; the fifth is an ID pin used to detect whether the connected peripheral is a host or a device.

The same transceiver chip (such as TUSB1105 from Texas Instruments) can be used on USB hosts and devices. You may thus choose to use the same transceiver part on buses 1 through 3 in Figure 11.2. OTG requires a special-purpose transceiver chip (such as ISP1301 from Philips Semiconductors), however.

Bus Speeds

USB supports three operational speeds. The original USB 1.0 specification supports 1.5MBps, referred to as low-speed USB. USB 1.1, the next version of the specification, handles 12MBps, called full-speed USB. The current level of the specification is USB 2.0, which supports 480MBps, or high-speed USB. USB 2.0 is backward-compatible with the earlier versions of the specification. Peripherals such as USB keyboards and mice are examples of low-speed devices, and USB storage drives are examples of full-speed and high-speed devices. Today's PC systems are USB 2.0-compliant and allow all three target speeds, but some embedded controllers adhere to USB 1.1 and support only full-speed and low-speed modes of operation.

Host Controllers

USB host controllers conform to one of a few standards:

- Universal Host Controller Interface (UHCI): The UHCI specification was initiated by Intel, so your PC is likely to have this controller if it's Intel-based.
- Open Host Controller Interface (OHCI): The OHCI specification originated from companies such as Compaq and Microsoft. An OHCI-compatible controller has more intelligence built in to hardware than UHCI, so an OHCI HCD is relatively simpler than a UHCI HCD.
- Enhanced Host Controller Interface (EHCI): This is the host controller that supports high-speed USB 2.0 devices. EHCI controllers usually have either a UHCI or OHCI companion controller to handle slower devices.
- USB OTG controllers: They are getting increasingly popular in embedded microcontrollers. With OTG support, each communicating end can act as a *dual-role device* (DRD). By initiating a dialog using the *Host Negotiation Protocol* (HNP), a DRD can switch itself to host mode or device mode based on the desired functionality.

In addition to these mainstream USB host controllers, Linux supports a few more controllers. An example is the HCD for the ISP116x chip.

Host controllers have a built-in hardware component called the *root hub.* The root hub is a virtual hub that sources USB ports. The ports, in turn, can connect to external or internal physical hubs and source more ports,

yielding a tree topology.

Transfer Types

Data exchange with a USB device can be one of four types:

- Control transfers, used to carry configuration and control information
- Bulk transfers that ferry large quantities of time-insensitive data
- Interrupt transfers that exchange small quantities of time-sensitive data
- Isochronous transfers for real-time data at predictable bit rates

A USB storage drive, for example, uses control transfers to issue disk access commands and bulk transfers to exchange data. A keyboard uses interrupt transfers to carry key strokes within predictable delays. A device that needs to stream audio data in real time uses isochronous transfers. The responsibilities of the four transfer types for USB Bluetooth devices are discussed in the section "Device Example: USB Adapter" in Chapter 16, "Linux Without Wires."

Addressing

Each addressable unit in a USB device is called an *endpoint.* The address assigned to an endpoint is called an *endpoint address.* Each endpoint address has an associated data transfer type. If an endpoint is responsible for bulk data transfer, for example, it's called a *bulk endpoint.* Endpoint address 0 is used exclusively for device configuration. A control pipe is attached to this endpoint for device enumeration (see the section "Enumeration").

An endpoint can be associated with upstream or downstream data transfer. Data arriving upstream from a device is called an `IN` transfer, whereas data flowing downstream to a device is an `OUT` transfer. `IN` and `OUT` transfers own separate address spaces. So, you can have a bulk `IN` endpoint and a bulk `OUT` endpoint answering to the same address.

USB resembles I^2C on some counts and PCI on others as summarized in Table 11.1. USB's device addressing is similar to I^2C, while it supports hotplugging like PCI. USB device addresses, like standard I^2C, do not consume a portion of the CPU's address space. Rather, they reside in a private space ranging from 1 to 127.

Table 11.1. USB's Similarities with I^2C and PCI

USB's similarities with I^2C:

- USB and I^2C are master-slave protocols.
- Device addresses reside in a private 7-bit space.
- Device-resident memory is not mapped to the CPU's memory or I/O space, so it does not consume CPU resources.

USB's similarities with PCI:

- Devices can be hotplugged.
- Device driver architecture resembles PCI drivers. Both classes of drivers own `probe()/disconnect()`[1] methods and ID tables identifying the devices they support.
- Supports high speeds. Slower than PCI, however. See Table 10.1 in Chapter 10, "Peripheral Component Interconnect," for the speeds supported by different members of the PCI family.
- USB host controllers, like PCI controllers, usually have built-in DMA engines that can master the bus.
- Supports multifunction devices. USB supports interface descriptors per function. Each PCI device function has its own device ID and configuration space.

[1] `disconnect()` is called `remove()` in PCI parlance.

Linux-USB Subsystem

Look at Figure 11.3 to understand the architecture of the Linux-USB subsystem. The constituent pieces of the subsystem are as follows:

- The USB core. Like the core layers of driver subsystems that you saw in previous chapters, the USB core is a code base consisting of routines and structures available to HCDs and client drivers. The core also provides a level of indirection that renders client drivers independent of host controllers.

- HCDs that drive different host controllers.

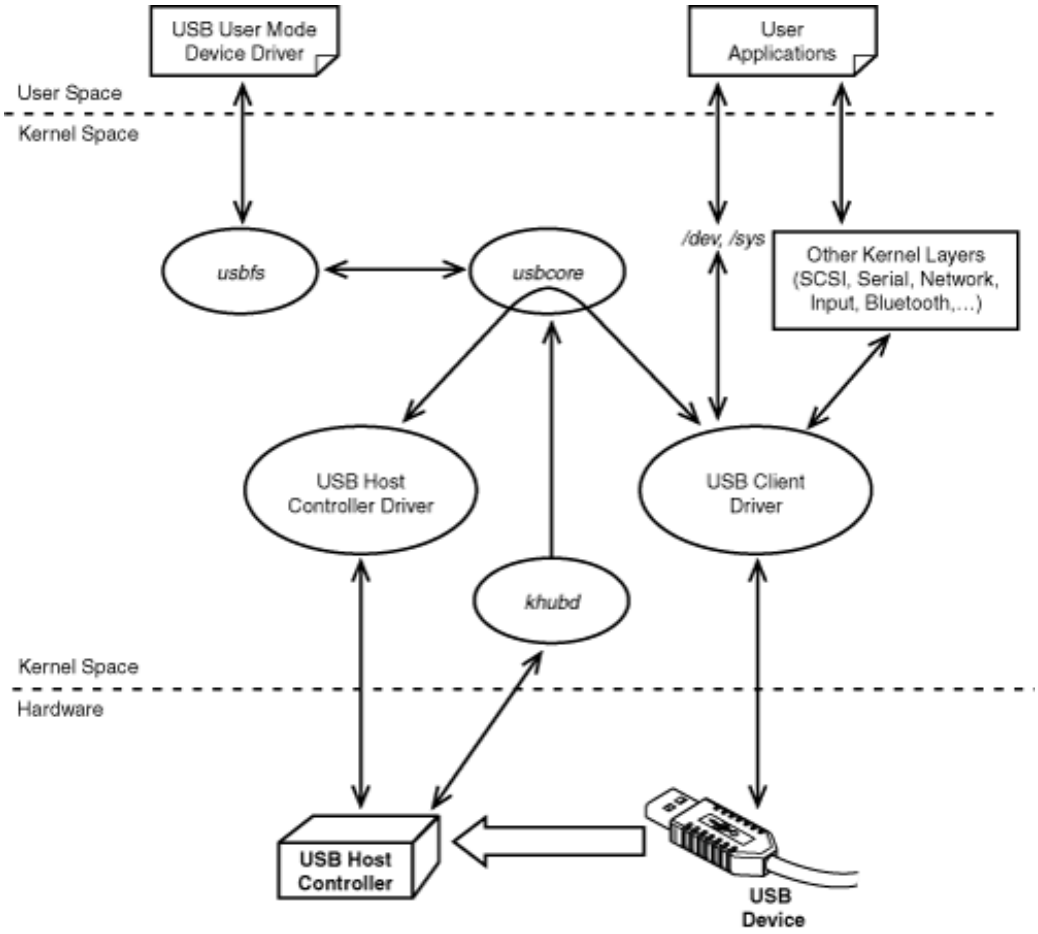
- A hub driver for the root hub (and physical hubs) and a helper kernel thread *khubd* that monitors all ports connected to the hub. Detecting port status changes and configuring hotplugged devices is time-consuming and is best accomplished using a helper thread for reasons you learned in Chapter 3, "Kernel Facilities." The khubd thread is asleep by default. The hub driver wakes khubd whenever it detects a status change on a USB port connected to it.

- Device drivers for USB client devices.

- The USB filesystem *usbfs* that lets you drive USB devices from user space. We discuss user mode USB drivers in Chapter 19, "Drivers in User Space."

Figure 11.3. The Linux-USB subsystem.

[View full size image]



For end-to-end operation, the USB subsystem calls on various other kernel layers for assistance. To support USB mass storage devices, for example, the USB subsystem works in tandem with SCSI drivers, as shown in Figure 11.3. To drive USB-Bluetooth keyboards, the stakeholders are fourfold: the USB subsystem, the Bluetooth layer, the input subsystem, and the tty layer.

Driver Data Structures

When you write a USB client driver, you have to work with several data structures. Let's look at the important ones.

The **usb_device** Structure

Each device driver subsystem relies on a special-purpose data structure to internally represent a device. The `usb_device` structure is to the USB subsystem, what `pci_dev` is to the PCI layer, and what `net_device` is to the network driver layer. `usb_device` is defined in *include/linux/usb.h* as follows:

```
struct usb_device {
  /* ... */
  enum usb_device_state state; /* Configured, Not Attached, etc */
  enum usb_device_speed speed; /* High/full/low (or error) */
  /* ... */
  struct usb_device *parent;   /* Our hub, unless we're the root */
  /* ... */
  struct usb_device_descriptor descriptor; /* Descriptor */
  struct usb_host_config *config;          /* All of the configs */
  struct usb_host_config *actconfig;       /* The active config */
  /* ... */
  int maxchild;                                /* No: of ports if hub */
  struct usb_device *children[USB_MAXCHILDREN]; /* Child devices */
  /* ... */
};
```

We use this structure when we develop an example driver for a USB telemetry card later.

USB Request Blocks

USB Request Block (URB) is the centerpiece of the USB data transfer mechanism. A URB is to the USB stack, what an `sk_buff` (discussed in Chapter 15, "Network Interface Cards") is to the networking stack.

Let's take a peek inside a URB. The following definition is from *include/linux/usb.h*, omitting fields not of particular interest to device drivers:

Code View:

```
struct urb
{
  struct kref kref;              /* Reference count of the URB */
  /* ... */
  struct usb_device *dev;        /* (in) pointer to associated
                                    device */
  unsigned int pipe;             /* (in) pipe information */
  int status;                    /* (return) non-ISO status */
  unsigned int transfer_flags;   /* (in) URB_SHORT_NOT_OK | ...*/
  void *transfer_buffer;         /* (in) associated data buffer */
  dma_addr_t transfer_dma;       /* (in) dma addr for
                                    transfer_buffer */
  int transfer_buffer_length;    /* (in) data buffer length */
```

```
  /* ... */
  unsigned char *setup_packet;   /* (in) setup packet */
  /* ... */
  int interval;                  /* (modify) transfer interval
                                    (INT/ISO) */
  /* ... */
  void *context;                 /* (in) context for completion */
  usb_complete_t complete;       /* (in) completion routine */
  /* ... */
};
```

There are three steps to using a URB: create, populate, and submit. To create a URB, use `usb_alloc_urb()`. This function allocates and zeros-out URB memory, initializes a kobject associated with the URB, and initializes a spinlock to protect the URB.

To populate a URB, use the following helper routines offered by the USB core:

```
void usb_fill_[control|int|bulk]_urb(
       struct urb *urb,              /* URB pointer */
       struct usb_device *usb_dev,   /* USB device structure */
       unsigned int pipe,            /* Pipe encoding */
       unsigned char *setup_packet,  /* For Control URBs only! */
       void *transfer_buffer,        /* Buffer for I/O */
       int buffer_length,            /* I/O buffer length */
       usb_complete_t completion_fn, /* Callback routine */
       void *context,                /* For use by completion_fn */
       int interval);                /* For Interrupt URBs only! */
```

The semantics of the previous routines will get clearer when we develop the example driver later on. These helper routines are available to control, interrupt, and bulk URBs but not to isochronous ones.

To submit a URB for data transfer, use `usb_submit_urb()`. URB submission is asynchronous. The `usb_fill_[control|int|bulk]_urb()` functions listed previously take the address of a callback function as argument. The callback routine executes after the URB submission completes and accomplishes things such as checking submission status and freeing the data-transfer buffer.

The USB core also offers wrapper interfaces that provide a façade of synchronous URB submission:

```
int usb_[control|interrupt|bulk]_msg(struct usb_device *usb_dev,
                                     unsigned int pipe, ...);
```

`usb_bulk_msg ()`, for example, builds a bulk URB, submits it, and blocks until the operation completes. You don't have to supply a callback function because a generic completion routine serves that purpose. You don't need to explicitly create and populate the URB either, because `usb_bulk_msg()` does that for you at no additional cost. We will use this interface in our example driver.

`usb_free_urb()` is used to free a reference to a completed URB, whereas `usb_unlink_urb()` cancels a pending URB operation.

As mentioned in the section "Sysfs, Kobjects, and Device Classes" in Chapter 4, "Laying the Groundwork," a URB contains a kref object to track references to it. `usb_submit_urb()` increments the reference count using `kref_get()`. `usb_free_urb()` decrements the reference count using `kref_put()` and performs the free

operation only if there are no remaining references.

A URB is associated with an abstraction called a *pipe*, which we discuss next.

Pipes

A pipe is an integer encoding of a combination of the following:

- The endpoint address
- The direction of data transfer (`IN` or `OUT`)
- The type of data transfer (control, interrupt, bulk, or isochronous)

A pipe is the address element of each USB data transfer and is an important field in the URB structure. To help populate this field, the USB core provides the following helper macros:

```
usb_[rcv|snd][ctrl|int|bulk|isoc]pipe(struct usb_device *usb_dev,
                                      __u8 endpointAddress);
```

where `usb_dev` is a pointer to the associated `usb_device` structure, and `endpointAddress` is the assigned endpoint address between 1 and 127. To create a bulk pipe in the `OUT` direction, for example, call `usb_sndbulkpipe()`. For a control pipe in the `IN` direction, invoke `usb_rcvctrlpipe()`.

While referring to a URB, it's often qualified by the transfer type of the associated pipe. If a URB is attached to a bulk pipe, for example, it's called a *bulk URB*.

Descriptor Structures

The USB specification defines a series of *descriptors* to hold information about a device. The Linux-USB core defines data structures corresponding to each descriptor. Descriptors are of four types:

- *Device descriptors* contain general information such as the product ID and vendor ID of the device. `usb_device_descriptor` is the structure corresponding to device descriptors.
- *Configuration descriptors* are used to describe different configuration modes such as bus-powered and self-powered operation. `usb_config_descriptor` is the data structure associated with configuration descriptors.
- *Interface descriptors* allow USB devices to support multiple functions. `usb_interface_descriptor` defines interface descriptors.
- *Endpoint descriptors* carry information associated with the final endpoints of a device. `usb_endpoint_descriptor` is the structure in question.

These descriptor formats are defined in Chapter 9 of the USB specification, whereas the matching structures are

defined in *include/linux/usb/ch9.h*. Listing 11.1 shows the hierarchical topology of the descriptors and prints all endpoint addresses associated with a USB device. To this end, it traverses the tree consisting of the four types of descriptors described previously. The following is the output generated by Listing 11.1 for a USB CD drive:

```
Endpoint Address = 1
Endpoint Address = 82
Endpoint Address = 83
```

The first address belongs to a bulk `IN` endpoint, the second address is owned by a bulk `OUT` endpoint, and the third addresses an interrupt `IN` endpoint.

There are more data structures associated with USB client drivers, such as `usb_device_id`, `usb_driver`, and `usb_class_driver`. We will meet them when we do hands-on development in the section "Device Example: Telemetry Card."

Listing 11.1. Print All USB Endpoint Addresses on a Device

[View full size image]

```
/* ... */
/* USB device */
struct usb_device *udevice;
/* ... */
struct usb_device_descriptor u_d_desc = udevice->descriptor;

/* Device's active configuration */
struct usb_host_config *uconfig;
struct usb_config_desriptor u_c_desc;

/* Interfaces in the active configuration */
struct usb_interface *uinterface;

/* Alternate Setting for this interface */
struct usb_host_interface *ualtsetting;
struct usb_interface_descriptor u_i_desc;

/* Endpoints for this altsetting */
struct usb_host_endpoint *uendpoint;
struct usb_endpoint_descriptor u_e_desc;

uconfig = udevice->actconfig;

u_c_desc = uconfig->desc;

for (i = 0; i < u_c_desc.bNumInterfaces;
     i++) {
  uinterface = udevice->actconfig->interface[i];
  for (j = 0; j < uinterface->num_altsetting; j++) {
ualtsetting = &uinterface->altsetting[j];
u_i_desc = ualtsetting->desc;
  for (k = 0; k < u_i_desc.bNumEndpoints; k++) {
    uendpoint = &ualtsetting->endpoint[k];
    u_e_desc = uendpoint->desc;
    printk ("Endpoint Address = %d\n",
            u_e_desc.bEndpointAddress\n");
  }
}
/* ... */
```

Enumeration

The life of a hotplugged USB device starts with a process called *enumeration* by which the host learns about the device's capabilities and configures it. The hub driver is the component in the Linux-USB subsystem responsible for enumeration. Let's look at the sequence of steps that achieve device enumeration when you plug in a USB pen drive into a host computer:

1. The root hub reports a change in the port's current due to the device attachment. The hub driver detects this status change, called a `USB_PORT_STAT_C_CONNECTION` in Linux-USB terminology, and awakens khubd.

2. Khubd deciphers the identity of the USB port subjected to the status change. In this case, it's the port where you plugged in the pen drive.

3. Next, khubd chooses a device address between 1 and 127 and assigns it to the pen drive's bulk endpoint using a control URB attached to endpoint `0`.

4. Khubd uses the above control URB attached to endpoint `0` to obtain the device descriptor from the pen drive. It then requests the device's configuration descriptors and selects a suitable one. In the case of the pen drive, only a single configuration descriptor is on offer.

5. Khubd requests the USB core to bind a matching client driver to the inserted device.

When enumeration is complete and the device is bound to a driver, khubd invokes the associated client driver's `probe()` method. In this case, khubd calls `storage_probe()` defined in *drivers/usb/storage/usb.c.* From this point on, the mass storage driver is responsible for normal device operation.

Device Example: Telemetry Card

Now that you know the basics of Linux-USB, it's time to look at an example device. Consider a system equipped with a telemetry card connected to the processor via internal USB, as shown in bus 2 of Figure 11.2. The card acquires data from a remote device and ferries it to the processor over USB. An example telemetry card is a medical-grade board that monitors or programs an implanted device.

Let's assume that our example telemetry card has the following endpoints having the semantics described in Table 11.2:

- A control endpoint attached to an on-card configuration register
- A bulk `IN` endpoint that passes remote telemetry information collected by the card to the processor
- A bulk `OUT` endpoint that transfers data in the reverse direction

Table 11.2. Register Space in the Telemetry Card

Register	Associated Endpoint
Telemetry Configuration Register	Control endpoint `0` (register offset `0xA`).
Telemetry Data-In Register	Bulk `IN` endpoint. The endpoint address is assigned during device enumeration.
Telemetry Data-Out Register	Bulk `OUT` endpoint. The endpoint address is assigned during device enumeration.

Let's build a minimal driver for this card partly based on the USB skeleton driver, *drivers/usb/usb-skeleton.c*.

Because PCMCIA, PCI, and USB devices have similar characteristics such as hotplug support, some driver methods and data structures belonging to these subsystems resemble each other. This is especially true for the portions responsible for initializing and probing. As we progress through the telemetry driver and notice parallels with what we learned for PCI drivers in Chapter 10, we will pause and take note.

Initializing and Probing

Like PCI and PCMCIA drivers, USB drivers have `probe()/disconnect()`[2] methods to support hotplugging, and a table that contains the identity of devices they support. A USB device is identified by the `usb_device_id` structure defined in *include/linux/mod_devicetable.h*. You may recall from the previous chapter that the `pci_device_id` structure, also defined in the same header file, identifies PCI devices.

[2] `disconnect()` is called `remove()` in PCI and PCMCIA parlance.

```
struct usb_device_id {
  /* ... */
  __u16         idVendor;        /* Vendor ID */
```

```
  __u16         idProduct;       /* Device ID */
  /* ... */
  __u8          bDeviceClass;    /* Device class */
  __u8          bDeviceSubClass; /* Device subclass */
  __u8          bDeviceProtocol; /* Device protocol */
 /* ... */
};
```

`idVendor` and `idProduct`, respectively, hold the manufacturer ID and product ID, whereas `bDeviceClass`, `bDeviceSubClass`, and `bDeviceProtocol` categorize the device based on its functionality. This classification, determined by the USB specification, allows implementation of generic client drivers as discussed in the section "Class Drivers" later.

Listing 11.2 implements the telemetry driver's initialization routine, `usb_tele_init()`, which calls on `usb_register()` to register its `usb_driver` structure with the USB core. As shown in the listing, `usb_driver` ties the driver's `probe()` method, `disconnect()` method, and `usb_device_id` table together. `usb_driver` is similar to `pci_driver`, except that the `disconnect()` method in the former is named `remove()` in the latter.

Listing 11.2. Initializing the Driver

Code View:

```
#define USB_TELE_VENDOR_ID    0xABCD  /* Manufacturer's Vendor ID */
#define USB_TELE_PRODUCT_ID   0xCDEF  /* Device's Product ID */

/* USB ID Table specifying the devices that this driver supports */
static struct usb_device_id tele_ids[] = {
  { USB_DEVICE(USB_TELE_VENDOR_ID, USB_TELE_PRODUCT_ID) },
  { } /* Terminate */
};

MODULE_DEVICE_TABLE(usb, tele_ids);

/* The usb_driver structure for this driver */
static struct usb_driver tele_driver
{
  .name       =  "tele",           /* Unique name */
  .probe      =  tele_probe,       /* See Listing 11.3 */
  .disconnect =  tele_disconnect,  /* See Listing 11.3 */
  .id_table   =  tele_ids,         /* See above */
};

/* Module Initialization */
static int __init
usb_tele_init(void)
{
  /* Register with the USB core */
  result = usb_register(&tele_driver);

  /* ... */
  return 0;
}

/* Module Exit */
static void __exit
usb_tele_exit(void)
{
  /* Unregister from the USB core */
```

```
  usb_deregister(&tele_driver);
  return;
}

module_init(usb_tele_init);
module_exit(usb_tele_exit);
```

The `USB_DEVICE()` macro creates a `usb_device_id` from the vendor and product IDs supplied to it. This is analogous to the `PCI_DEVICE()` macro discussed in the previous chapter. The `MODULE_DEVICE_TABLE()` macro marks `tele_ids` in the module image so that the module can be loaded on demand if the card is hotplugged. This is again similar to what we discussed for PCMCIA and PCI devices in the previous two chapters.

When the USB core detects a device with properties matching the ones declared in the `usb_device_id` table belonging to a client driver, it invokes the `probe()` method registered by that driver. When the device is unplugged or if the module is unloaded, the USB core invokes the driver's `disconnect()` method.

Listing 11.3 implements the `probe()` and `disconnect()` methods of the telemetry driver. It starts by defining a device-specific structure, `tele_device_t`, which contains the following fields:

- A pointer to the associated `usb_device`.
- A pointer to the `usb_interface`. Revisit Listing 11.1 to see this structure in use.
- A control URB (`ctrl_urb`) to communicate with the telemetry configuration register, and a `ctrl_req` to formulate programming requests to this register. These fields are described in the next section "Accessing Registers."
- The card has a bulk `IN` endpoint through which you can glean the collected telemetry information. Associated with this endpoint are three fields: `bulk_in_addr`, which holds the endpoint address; `bulk_in_buf`, which stores received data; and `bulk_in_len`, which contains the size of the receive data buffer.
- The card has a bulk `OUT` endpoint to facilitate downstream data transfer. `tele_device_t` has a field called `bulk_out_addr` to store the address of this endpoint. There are fewer data structures in the `OUT` direction because in this simple case we use a synchronous URB submission interface that hides several implementation details.

Khubd invokes the card's `probe()` method, `tele_probe()`, soon after enumeration. `tele_probe()` performs three tasks:

1. Allocates memory for the device-specific structure, `tele_device_t`.

2. Initializes the following fields in `tele_device_t` related to the device's bulk endpoints: `bulk_in_buf`, `bulk_in_len`, `bulk_in_addr`, and `bulk_out_addr`. For this, it uses the data collected by the hub driver during enumeration. This data is available in descriptor structures discussed in the section "Descriptor Structures."

3. Exports the character device */dev/tele* to user space. Applications operate over */dev/tele* to exchange data with the telemetry card. `tele_probe()` invokes `usb_register_dev()` and supplies it the `file_operations` that form the underlying pillars of the */dev/tele* interface via the `usb_class_driver` structure.

The address of the device-specific structure allocated in Step 1 has to be saved so that other methods can access it. To achieve this, the telemetry driver uses a threefold strategy depending on the function arguments available to various driver routines. To save this structure pointer between the `probe()` and `open()` invocation threads, the driver uses the device's `driver_data` field via the pair of functions, `usb_set_intfdata()` and `usb_get_intfdata()`. To save the address of the structure pointer between the `open()` thread and other entry points, `open()` stores it in the */dev/tele*'s `file->private_data` field. This is because the kernel supplies these char entry points with */dev/tele*'s `inode` pointer as argument rather than the `usb_interface` pointer. To glean the address of the device-specific structure from URB callback functions, the associated submission threads use the URB's `context` field as the storage area. Look at Listings 11.3, 11.4, and 11.5 to see these mechanisms in action.

All USB character devices answer to major number `180`. If you enable `CONFIG_USB_DYNAMIC_MINORS` during kernel configuration, the USB core dynamically selects a minor number from the available pool. This behavior is similar to registering misc drivers after specifying `MISC_DYNAMIC_MINOR` in the `miscdevice` structure (as discussed in the section "Misc Drivers" in Chapter 5, "Character Drivers"). If you choose not to enable `CONFIG_USB_DYNAMIC_MINORS`, the USB subsystem selects an available minor number starting at the minor base set in the `usb_class_driver` structure.

Listing 11.3. Probing and Disconnecting

Code View:

```
/* Device-specific structure */
typedef struct {
  struct usb_device      *usbdev;         /* Device representation */
  struct usb_interface   *interface;      /* Interface representation*/
  struct urb             *ctrl_urb;       /* Control URB for
                                             register access */
  struct usb_ctrlrequest  ctrl_req;       /* Control request
                                             as per the spec */
  unsigned char          *bulk_in_buf;    /* Receive data buffer */
  size_t                  bulk_in_len;    /* Receive buffer size */
  __u8                    bulk_in_addr;   /* IN endpoint address */
  __u8                    bulk_out_addr;  /* OUT endpoint address */
  /* ... */                               /* Locks, waitqueues,
                                             statistics.. */
} tele_device_t;

#define TELE_MINOR_BASE 0xAB    /* Assigned by the Linux-USB
                                   subsystem maintainer */

/* Conventional char driver entry points.
   See Chapter 5, "Character Drivers." */
static struct file_operations tele_fops =
{
  .owner   =  THIS_MODULE,  /* Owner */
```

```
  .read    =  tele_read,    /* Read method */
  .write   =  tele_write,   /* Write method */
  .ioctl   =  tele_ioctl,   /* Ioctl method */
  .open    =  tele_open,    /* Open method */
  .release =  tele_release, /* Close method */
};


static struct usb_class_driver tele_class = {
  .name       = "tele",
  .fops       = &tele_fops,      /* Connect with /dev/tele */
  .minor_base = TELE_MINOR_BASE, /* Minor number start */
};
/* The probe() method is invoked by khubd after device
   enumeration. The first argument, interface, contains information
   gleaned during the enumeration process. id is the entry in the
   driver's usb_device_id table that matches the values read from
   the telemetry card. tele_probe() is based on skel_probe()
   defined in drivers/usb/usb-skeleton.c */
static int
tele_probe(struct usb_interface *interface,
           const struct usb_device_id *id)
{
  struct usb_host_interface *iface_desc;
  struct usb_endpoint_descriptor *endpoint;
  tele_device_t *tele_device;
  int retval = -ENOMEM;

  /* Allocate the device-specific structure */
  tele_device = kzalloc(sizeof(tele_device_t), GFP_KERNEL);

  /* Fill the usb_device and usb_interface */
  tele_device->usbdev =
                usb_get_dev(interface_to_usbdev(interface));
  tele_device->interface = interface;

  /* Set up endpoint information from the data gleaned
     during device enumeration */
  iface_desc = interface->cur_altsetting;
  for (int i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;

    if (!tele_device->bulk_in_addr &&
        usb_endpoint_is_bulk_in(endpoint)) {
      /* Bulk IN endpoint */
      tele_device->bulk_in_len =
                      le16_to_cpu(endpoint->wMaxPacketSize);
      tele_device->bulk_in_addr = endpoint->bEndpointAddress;
      tele_device->bulk_in_buf =
                kmalloc(tele_device->bulk_in_len, GFP_KERNEL);
    }

    if (!tele_device->bulk_out_addr &&
       usb_endpoint_is_bulk_out(endpoint)) {
      /* Bulk OUT endpoint */
      tele_device->bulk_out_addr = endpoint->bEndpointAddress;
    }
   }
```

```
   if (!(tele_device->bulk_in_addr && tele_device->bulk_out_addr)) {
     return retval;
   }

   /* Attach the device-specific structure to this interface.
      We will retrieve it from tele_open() */
   usb_set_intfdata(interface, tele_device);

   /* Register the device */
   retval = usb_register_dev(interface, &tele_class);
   if (retval) {
     usb_set_intfdata(interface, NULL);
     return retval;
   }

   printk("Telemetry device now attached to /dev/tele\n");
   return 0;
}

/* Disconnect method. Called when the device is unplugged or when the module is
   unloaded */
static void
tele_disconnect(struct usb_interface *interface)
{
  tele_device_t *tele_device;
  /* ... */

  /* Reverse of usb_set_intfdata() invoked from tele_probe() */
  tele_device = usb_get_intfdata(interface);

  /* Zero out interface data */
  usb_set_intfdata(interface, NULL);

  /* Release /dev/tele */
  usb_deregister_dev(interface, &tele_class);

  /* NULL the interface. In the real world, protect this
     operation using locks */
  tele_device->interface = NULL;
  /* ... */
}
```

Accessing Registers

The `open()` method initializes the on-card telemetry configuration register when an application opens */dev/tele*. To set the contents of this register, `tele_open()` submits a control URB attached to the default endpoint `0`. When you submit a control URB, you have to supply an associated control request. The structure that sends a control request to a USB device has to conform to Chapter 9 of the USB specification and is defined as follows in *include/linux/usb/ch9.h*:

```
struct usb_ctrlrequest {
  __u8 bRequestType;
  __u8 bRequest;
```

```
  __le16 wValue;
  __le16 wIndex;
  __le16 wLength;
} __attribute__ ((packed));
```

Let's take a look at the components that make up a `usb_ctrlrequest`. The `bRequest` field identifies the control request. `bRequestType` qualifies the request by encoding the data transfer direction, the request category, and whether the recipient is a device, interface, endpoint, or something else. `bRequest` can either belong to a set of standard values or be vendor-defined. In our example, the `bRequest` for writing to the telemetry configuration register is a vendor-defined one. `wValue` holds the data to be written to the register, `wIndex` is the desired offset into the register space, and `wLength` is the number of bytes to be transferred.

Listing 11.4 implements `tele_open()`. Its main task is to program the telemetry configuration register with appropriate values. Before browsing the listing, revisit the `tele_device_t` structure defined in Listing 11.3 focusing on two fields: `ctrl_urb` and `ctrl_req`. The former is a control URB for communicating with the configuration register, whereas the latter is the associated `usb_ctrlrequest`.

To program the telemetry configuration register, `tele_open()` does the following:

1. Allocates a control URB to prepare for the register write.

2. Creates a `usb_ctrlrequest` and populates it with the request identifier, request type, register offset, and the value to be programmed.

3. Creates a control pipe attached to endpoint `0` of the telemetry card to carry the control URB.

4. Because `tele_open()` submits the URB asynchronously, it needs to wait for the associated callback function to finish before returning to its caller. To this end, `tele_open()` calls on the kernel's completion API for assistance using `init_completion()`. Step 7 calls the matching `wait_for_completion()` that waits until the callback function invokes `complete()`. We discussed the completion API in the section "Completion Interface" in Chapter 3.

5. Initializes fields in the control URB using `usb_fill_control_urb()`. This includes the `usb_ctrlrequest` populated in Step 2.

6. Submits the URB to the USB core using `usb_submit_urb()`.

7. Waits until the callback function signals that the register programming is complete.

8. Returns the status.

Listing 11.4. Initialize the Telemetry Configuration Register

Code View:

```
/* Offset of the Telemetry configuration register
   within the on-card register space */
#define TELEMETRY_CONFIG_REG_OFFSET      0x0A

/* Value to program in the configuration register */
#define TELEMETRY_CONFIG_REG_VALUE       0xBC

/* The vendor-defined bRequest for programming the
   configuration register */
```

```
#define TELEMETRY_REQUEST_WRITE          0x0D

/* The vendor-defined bRequestType */
#define TELEMETRY_REQUEST_WRITE_REGISTER 0x0E

/* Open method */
static int
tele_open(struct inode *inode, struct file *file)
{
  struct completion tele_config_done;
  tele_device_t *tele_device;
  void *tele_ctrl_context;
  char *tmp;
  __le16 tele_config_index = TELEMETRY_CONFIG_REG_OFFSET;
  unsigned int tele_ctrl_pipe;
  struct usb_interface *interface;

  /* Obtain the pointer to the device-specific structure.
     We saved it using usb_set_intfdata() in tele_probe() */
  interface = usb_find_interface(&tele_driver, iminor(inode));
  tele_device = usb_get_intfdata(interface);

  /* Allocate a URB for the control transfer */
  tele_device->ctrl_urb = usb_alloc_urb(0, GFP_KERNEL);
  if (!tele_device->ctrl_urb) {
    return -EIO;
  }

  /* Populate the Control Request */
  tele_device->ctrl_req.bRequestType = TELEMETRY_REQUEST_WRITE;
  tele_device->ctrl_req.bRequest =
                        TELEMETRY_REQUEST_WRITE_REGISTER;
  tele_device->ctrl_req.wValue =
                        cpu_to_le16(TELEMETRY_CONFIG_REG_VALUE);
  tele_device->ctrl_req.wIndex =
                        cpu_to_le16p(&tele_config_index);
  tele_device->ctrl_req.wLength = cpu_to_le16(1);
  tele_device->ctrl_urb->transfer_buffer_length = 1;
  tmp = kmalloc(1, GFP_KERNEL);
  *tmp = TELEMETRY_CONFIG_REG_VALUE;

  /* Create a control pipe attached to endpoint 0 */
  tele_ctrl_pipe = usb_sndctrlpipe(tele_device->usbdev, 0);

  /* Initialize the completion mechanism */
  init_completion(&tele_config_done);

  /* Set URB context. The context is part of the URB that is passed
     to the callback function as an argument. In this case, the
     context is the completion structure, tele_config_done */
  tele_ctrl_context = (void *)&tele_config_done;

  /* Initialize the fields in the control URB */
  usb_fill_control_urb(tele_device->ctrl_urb, tele_device->usbdev,
                       tele_ctrl_pipe,
                       (char *) &tele_device->ctrl_req,
                       tmp, 1, tele_ctrl_callback,
                       tele_ctrl_context);
  /* Submit the URB */
```

```
  usb_submit_urb(tele_device->ctrl_urb, GFP_ATOMIC);

  /* Wait until the callback returns indicating that the telemetry
     configuration register has been successfully initialized */
  wait_for_completion(&tele_config_done);

  /* Release our reference to the URB */
  usb_free_urb(urb);

  kfree(tmp);

  /* Save the device-specific object to the file's private_data
     so that you can directly retrieve it from other entry points
     such as tele_read() and tele_write() */
  file->private_data = tele_device;

  /* Return the URB transfer status */
  return(tele_device->ctrl_urb->status);
}


/* Callback function */
static void
tele_ctrl_callback(struct urb *urb)
{
  complete((struct completion *)urb->context);
}
```

You can render `tele_open()` simpler using `usb_control_msg()`, a blocking version of `usb_submit_urb()` that internally hides synchronization and callback details for control URBs. We preferred the asynchronous approach for learning purposes.

Data Transfer

Listing 11.5 implements the `read()` and `write()` entry points of the telemetry driver. These methods perform the real work when an application reads or writes to */dev/tele*. `tele_read()` performs synchronous URB submission because the calling process wants to block until telemetry data is available. `tele_write()`, however, uses asynchronous submission and returns to the calling thread without waiting for a confirmation that the data accepted by the driver has been successfully transferred to the device.

Because asynchronous transfers go hand in hand with a callback routine, Listing 11.5 implements `tele_write_callback()`. This routine examines `urb->status` to decipher the submission status. It also frees the transfer buffer allocated by `tele_write()`.

Listing 11.5. Data Exchange with the Telemetry Card

```
Code View:
/* Read entry point */
static ssize_t
tele_read(struct file *file, char *buffer,
          size_t count, loff_t *ppos)
{
```

```
  int retval, bytes_read;
  tele_device_t *tele_device;

  /* Get the address of tele_device */
  tele_device = (tele_device_t *)file->private_data;

  /* ... */

  /* Synchronous read */
  retval = usb_bulk_msg(tele_device->usbdev, /* usb_device */
       usb_rcvbulkpipe(tele_device->usbdev,
                tele_device->bulk_in_addr),  /* Pipe */
       tele_device->bulk_in_buf,             /* Read buffer */
       min(tele_device->bulk_in_len, count), /* Bytes to read */
       &bytes_read,                          /* Bytes read */
       5000);                                /* Timeout in 5 sec */

  /* Copy telemetry data to user space */
  if (!retval) {
    if (copy_to_user(buffer, tele_device->bulk_in_buf,
                     bytes_read)) {
      return -EFAULT;
    } else {
      return bytes_read;
    }
  }
  return retval;
}

/* Write entry point */
static ssize_t
tele_write(struct file *file, const char *buffer,
           size_t write_count, loff_t *ppos)
{
  char *tele_buf = NULL;
  struct urb *urb = NULL;
  tele_device_t *tele_device;

  /* Get the address of tele_device */
  tele_device = (tele_device_t *)file->private_data;

  /* ... */

  /* Allocate a bulk URB */
  urb = usb_alloc_urb(0, GFP_KERNEL);
  if (!urb) {
    return -ENOMEM;
  }

  /* Allocate a DMA-consistent transfer buffer and copy in
     data from user space. On return, tele_buf contains
     the buffer's CPU address, while urb->transfer_dma
     contains the DMA address */
  tele_buf = usb_buffer_alloc(tele_dev->usbdev, write_count,
                              GFP_KERNEL, &urb->transfer_dma);
  if (copy_from_user(tele_buf, buffer, write_count)) {
    usb_buffer_free(tele_device->usbdev, write_count,
                    tele_buf, urb->transfer_dma);
    usb_free_urb(urb);
```

```
    return -EFAULT
  }

  /* Populate bulk URB fields */
  usb_fill_bulk_urb(urb, tele_device->usbdev,
                    usb_sndbulkpipe(tele_device->usbdev,
                    tele_device->bulk_out_addr),
                    tele_buf, write_count, tele_write_callback,
                    tele_device);
  /* urb->transfer_dma is valid, so preferably utilize
     that for data transfer */
  urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

  /* Submit URB asynchronously */
  usb_submit_urb(urb, GFP_KERNEL);
  /* Release URB reference */
  usb_free_urb(urb);

  return(write_count);
}

/* Write callback */
static void
tele_write_callback(struct urb *urb)
{
  tele_device_t *tele_device;

  /* Get the address of tele_device */
  tele_device = (tele_device_t *)urb->context;

  /* urb->status contains the submission status. It's 0 if
     successful. Resubmit the URB in case of errors other than
     -ENOENT, -ECONNRESET, and -ESHUTDOWN */
  /* ... */

  /* Free the transfer buffer. usb_buffer_free() is the
     release-counterpart of usb_buffer_alloc() called
     from tele_write() */
  usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                  urb->transfer_buffer, urb->transfer_dma);

}
```

Class Drivers

The USB specification introduces the concept of device classes and describes the functionality of each class driver. Examples of standard device classes include mass storage, networking, hubs, serial converters, audio, video, imaging, modems, printers, and *human interface devices* (HIDs). Class drivers are generic and let you plug and play a wide array of cards without the need for developing and installing drivers for every single device. The Linux-USB subsystem includes support for major class drivers.

Each USB device has a class and a subclass code. The mass storage class (`0x08`), for example, supports subclasses such as compact disc (`0x02`), tape (`0x03`), and solid-state storage (`0x06`). As you saw previously, device drivers populate the `usb_device_id` structure with the classes and subclasses they support. You can glean a device's class and subclass information by looking at the "I:" lines in the */proc/bus/usb/devices* output.

Let's take a look at some important class drivers.

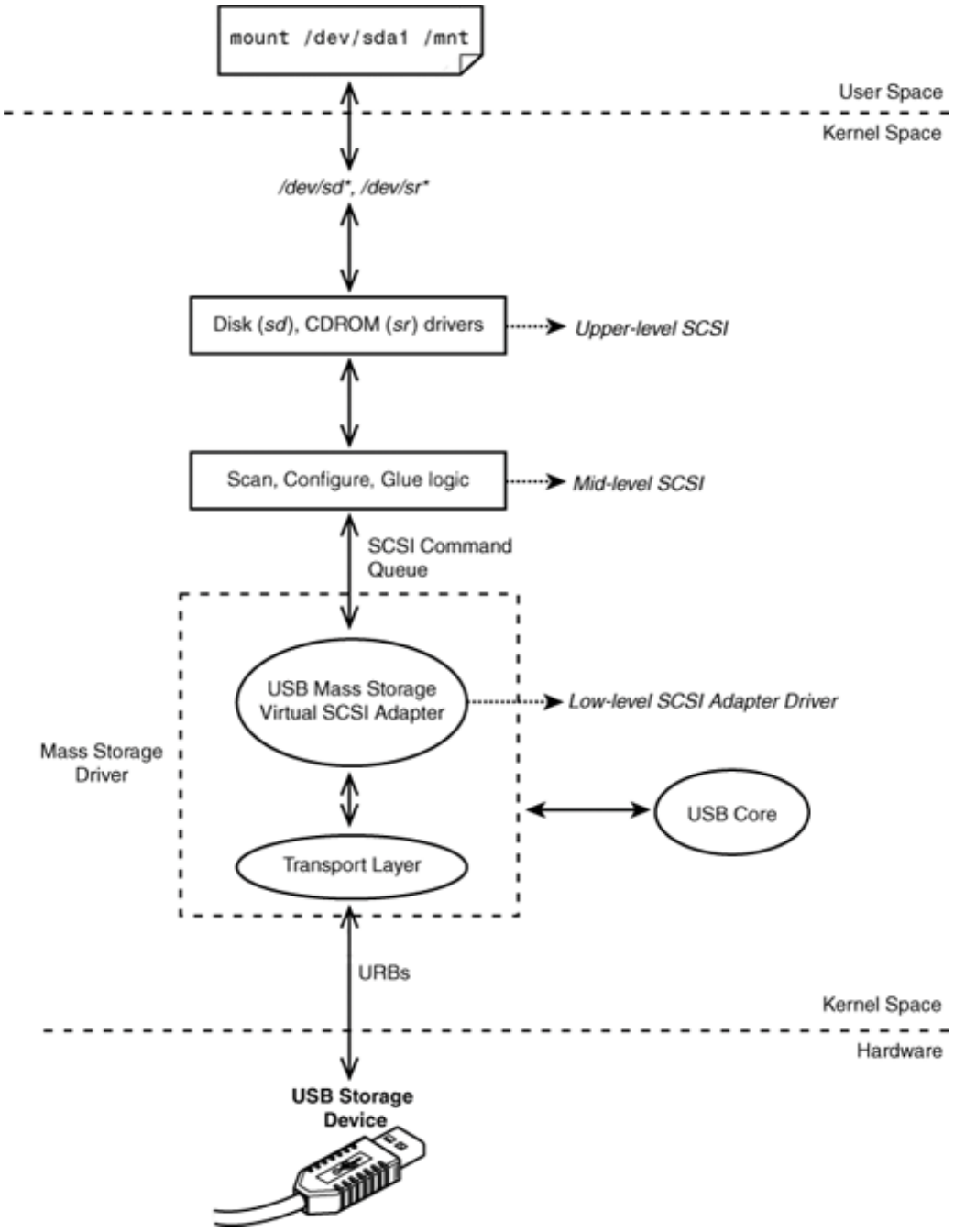
Mass Storage

In USB parlance, mass storage refers to USB hard disks, pen drives, CD-ROMs, floppy drives, and similar storage devices. USB mass storage devices adhere to the *Small Computer System Interface* (SCSI) protocol to communicate with host systems. Block access to USB storage devices is hence routed through the kernel's SCSI subsystem. Figure 11.4 provides you an overview of the interaction between USB storage and SCSI subsystems. As shown in the figure, the SCSI subsystem is architected into three layers:

1. Top-level drivers for devices such as disks (*sd.c*) and CD-ROMs (*sr.c*)
2. A middle-level layer that scans the bus, configures devices, and glues top-level drivers to low-level drivers
3. Low-level SCSI adapter drivers

Figure 11.4. USB mass storage and SCSI.

[View full size image]



The mass storage driver registers itself as a virtual SCSI adapter. The virtual adapter communicates upstream via SCSI commands and downstream using URBs. A USB disk appears to higher layers as a SCSI device attached to this virtual adapter.

To better understand the interactions between the USB and SCSI layers, let's implement a modification to the USB mass storage driver. The *usbfs* node */proc/bus/usb/devices*, contains the properties and connection details of all USB devices attached to the system. The "T:" line in the */proc/bus/usb/devices* output, for example, contains the bus number, the device's depth from the root hub, operational speed, and so on. The "P:" line contains the vendor ID, product ID, and revision number of the device. All the information available in */proc/bus/usb/devices* is managed by the USB subsystem, but there is one piece missing that is under the jurisdiction of the SCSI subsystem. The */dev* node name associated with the USB storage device (*sd[a-z][1-9]*

for disks and *sr[0-9]* for CD-ROMs) is not available in */proc/bus/usb/devices.* To overcome this limitation, let's add an "N:" line that displays the */dev* node name associated with the device. Listing 11.6 shows the necessary code changes in the form of a source patch to the 2.6.23.1 kernel tree.

Listing 11.6. Adding a Disk's */dev* Name to usbfs

Code View:

***include/scsi/scsi_host.h*:**

```
struct Scsi_Host {
  /* ... */
  void *shost_data;
+ char snam[8];      /* /dev node name for this disk */
  /* ... */
};
```

***drivers/usb/storage/usb.h*:**

```
struct us_data {
  /* ... */
+ char magic[4];
};
```

***include/linux/usb.h*:**

```
struct usb_interface {
  /* ... */
+ void *private_data;
};
```

***drivers/usb/storage/usb.c*:**

```
static int storage_probe(struct usb_interface *intf,
                         const struct usb_device_id *id)
{
  /* ... */
  memset(us, 0, sizeof(struct us_data));
+ intf->private_data = (void *) us;
+ strncpy(us->magic, "disk", 4);
  mutex_init(&(us->dev_mutex));
  /* ... */
}
```

***drivers/scsi/sd.c*:**

```
static int sd_probe(struct device *dev)
{
  /* ... */
  add_disk(gd);
+ memset(sdp->host->snam,0, sizeof(sdp->host->snam));
+ strncpy(sdp->host->snam, gd->disk_name, 3);
  sdev_printk(KERN_NOTICE, sdp, "Attached scsi %sdisk %s\n",
              sdp->removable ? "removable " : "", gd->disk_name);
  /* ... */
}
```

***drivers/scsi/sr.c*:**

```
static int sr_probe(struct device *dev)
{
```

```
  /* ... */
  add_disk(disk);
+ memset(sdev->host->snam,0, sizeof(sdev->host->snam));
+ strncpy(sdev->host->snam, cd->cdi.name, 3);
  sdev_printk(KERN_DEBUG, sdev, "Attached scsi CD-ROM %s\n",
              cd->cdi.name);
  /* ... */
}

drivers/usb/core/devices.c:
  /* ... */
  #include <asm/uaccess.h>
+ #include <scsi/scsi_host.h>
+ #include "../storage/usb.h"

static ssize_t usb_device_dump(char __user **buffer, size_t *nbytes,
                        loff_t *skip_bytes, loff_t *file_offset,
                        struct usb_device *usbdev,
                        struct usb_bus *bus, int level,
                        int index, int count)
{
  /* ... */
  ssize_t total_written = 0;
+ struct us_data *us_d;
+ struct Scsi_Host *s_h;
  /* ... */
  data_end = pages_start + sprintf(pages_start, format_topo,
                                   bus->busnum, level,
                                   parent_devnum,
                                   index, count, usbdev->devnum,
                                   speed, usbdev->maxchild);
+ /* Assume this device supports only one interface */
+ us_d =  (struct us_data *)
+           (usbdev->actconfig->interface[0]->private_data);
+
+ if ((us_d) && (!strncmp(us_d->magic, "disk", 4))) {
+   s_h = (struct Scsi_Host *) container_of((void *)us_d,
+                                           struct Scsi_Host,
+                                           hostdata);
+   data_end += sprintf(data_end, "N: ");
+   data_end += sprintf(data_end, "Device=%.100s",s_h->snam);
+   if (!strncmp(s_h->snam, "sr", 2)) {
+     data_end += sprintf(data_end, " (CDROM)\n");
+   } else if (!strncmp(s_h->snam, "sd", 2)) {
+     data_end += sprintf(data_end, " (Disk)\n");
+   }
+ }
   /* ... */
}
```

To understand Listing 11.6, let's first trace the code flow, continuing from where we left off in the section "Enumeration." In that section, we inserted a USB pen drive and followed the execution train until the invocation of `storage_probe()`, the `probe()` method of the mass storage driver. Moving further:

1. `storage_probe()` registers a virtual SCSI adapter by calling `scsi_add_host()`, supplying a private data structure called `us_data` as argument. `scsi_add_host()` returns a `Scsi_Host` structure for this virtual adapter, with space at the end for `us_data`.

2. It starts a kernel thread called *usb-storage* to handle all SCSI commands queued to the virtual adapter.

3. It schedules a kernel thread called *usb-stor-scan* that requests the SCSI middle-level layer to scan the bus for attached devices.

4. The device scan initiated in Step 3 discovers the presence of the inserted pen drive and binds the upper-level SCSI disk driver (*sd.c*) to the device. This results in the invocation of the SCSI disk driver's probe method, `sd_probe()`.

5. The *sd* driver allocates a */dev/sd** node to the disk. From this point on, applications use this interface to access the USB disk. The SCSI subsystem queues disk commands to the virtual adapter, which the usb-storage kernel thread handles using appropriate URBs.

Now that you know the basics, let's dissect Listing 11.6, looking at the data structure additions first. The listing adds a `snam` field to the `Scsi_Host` structure to hold the associated SCSI */dev* name that we are interested in. It also adds a private field to the `usb_interface` structure to associate each USB interface with its `us_data`. Because `us_data` is relevant only for storage devices, we need to ensure the validity of the private field of a USB interface before accessing it as `us_data`. For this, Listing 11.6 adds a magic string, "disk," to `us_data`.

The *usbfs* modification in Listing 11.6 (to *drivers/usb/core/devices.c*) pulls out the `us_data` associated with each interface via the private data field of its `usb_interface`. It then latches on to the associated `Scsi_Host` using the `container_of()` function, because as you saw in Step 1 previously, `usb_data` is glued to the end of the corresponding `Scsi_Host`. As you further saw in Step 5, `Scsi_Host` contains the */dev* node names that the sd and sr drivers populate. Usbfs stitches together an "N:" line using this information.

The following is the */proc/bus/usb/devices* output after integrating the changes in Listing 11.6 and attaching a PNY USB pen drive, an Addonics CD-ROM drive, and a Seagate hard disk to a laptop via a USB hub. The "N:" lines announce the identity of the */dev* node corresponding to each device:

Code View:

```
bash> cat /proc/bus/usb/devices
...
T:  Bus=04 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#= 3 Spd=480 MxCh= 0
N:  Device=sda(Disk)
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P:  Vendor=154b ProdID=0002 Rev= 1.00
S:  Manufacturer=PNY
S:  Product=USB 2.0 FD
S:  SerialNumber=6E5C07005B4F
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-
    storage
E:  Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms

T:  Bus=04 Lev=02 Prnt=02 Port=01 Cnt=02 Dev#= 5 Spd=480 MxCh= 0
N:  Device=sr0(CDROM)
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P:  Vendor=0bf6 ProdID=a002 Rev= 3.00
S:  Manufacturer=Addonics
```

```
S:  Product=USB to IDE Cable
S:  SerialNumber=1301011002A9AFA9
C:* #Ifs= 1 Cfg#= 2 Atr=c0 MxPwr= 98mA
I:* If#= 0 Alt= 0 #EPs= 3 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-
    storage
E:  Ad=01(O) Atr=02(Bulk) MxPS= 512 Ivl=125us
E:  Ad=82(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=83(I) Atr=03(Int.) MxPS=   2 Ivl=32ms

T:  Bus=04 Lev=02 Prnt=02 Port=02 Cnt=03 Dev#= 4 Spd=480 MxCh= 0
N:  Device=sdb(Disk)
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P:  Vendor=0bc2 ProdID=0501 Rev= 0.01
S:  Manufacturer=Seagate
S:  Product=USB Mass Storage
S:  SerialNumber=000000062459
C:* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr=  0mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-
    storage
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=88(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
...
```

As you can see, the SCSI subsystem has allotted *sda* to the pen drive, *sr0* to the CD-ROM, and *sdb* to the hard disk. User-space applications operate on these nodes to communicate with the respective devices. As you saw in Chapter 4, with the arrival of udev, however, you have the option of creating higher-level abstractions to identify each device without relying on the identity of the */dev* names allocated by the SCSI subsystem.

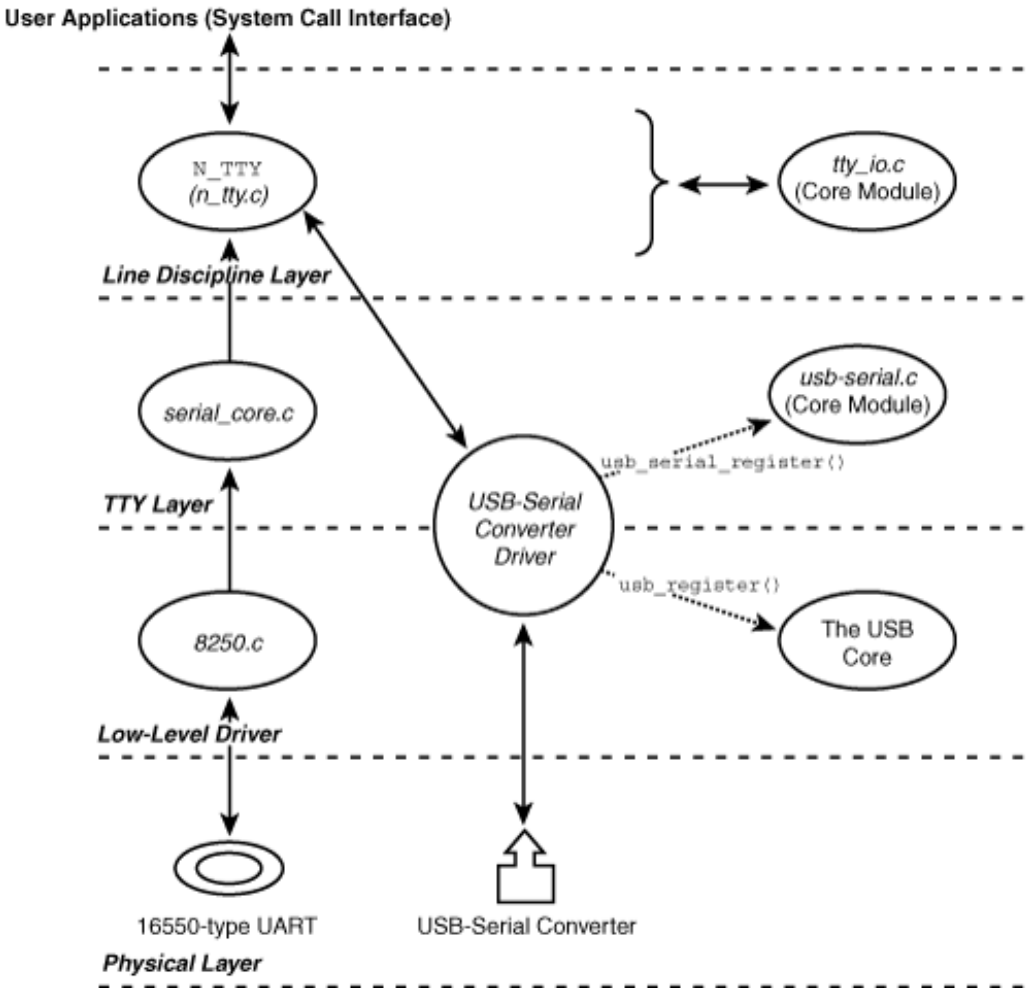
USB-Serial

USB-to-serial converters bring serial port capabilities to your computer via USB. You can use a USB-to-serial converter, for example, to get a serial debug console from an embedded device on a development laptop that has no serial ports.

In Chapter 6, "Serial Drivers," you learned the benefits of the kernel's layered serial architecture. Figure 11.5 illustrates how the USB-Serial layer fits into the kernel's serial framework.

Figure 11.5. The USB-Serial layer.

[View full size image]



A USB-serial driver is similar to other USB client drivers except that it avails the services of a USB-Serial core in addition to the USB core. The USB-Serial core provides the following:

- A tty driver that insulates low-level USB-to-serial converter drivers from higher serial layers such as line disciplines.
- Generic `probe()` and `disconnect()` routines that individual USB-serial drivers can leverage.
- Device nodes to access USB-serial ports from user space. Applications operate on USB-serial ports via */dev/ttyUSBX*, where *X* is the serial port number. Terminal emulators such as *minicom* and protocols such as PPP run unchanged over these interfaces.

A low-level USB-to-serial converter driver essentially does the following:

1. Registers a `usb_serial_driver` structure with the USB-Serial core using `usb_serial_register()`. The entry points supplied as part of `usb_serial_driver` form the crux of the driver.

2. Populates a `usb_driver` structure and registers it with the USB core using `usb_register()`. This is similar to what the example telemetry driver does, except that a serial converter driver can count on the generic `probe()` and `disconnect()` routines provided by the USB-Serial core.

Listing 11.7 contains snippets from the FTDI driver (*drivers/usb/serial/ftdi_sio.c*) that accomplish these two registrations for USB-to-serial converters based on FTDI chipsets.

Listing 11.7. A Snippet from the FTDI Driver

Code View:

```
/* The usb_driver structure */
static struct usb_driver ftdi_driver = {
  .name          =  "ftdi_sio",           /* Name */
  .probe         =  usb_serial_probe,     /* Provided by the
                                             USB-Serial core */
  .disconnect    =  usb_serial_disconnect,/* Provided by the
                                             USB-Serial core */
  .id_table      =  id_table_combined,    /* List of supported
                                             devices built
                                             around the FTDI chip */
  .no_dynamic_id =  1,                    /* Supported ids cannot be
                                             added dynamically */
};

/* The usb_serial_driver structure */
static struct usb_serial_driver ftdi_sio_device = {
  /* ... */
  .num_ports           = 1,
  .probe               = ftdi_sio_probe,
  .port_probe          = ftdi_sio_port_probe,
  .port_remove         = ftdi_sio_port_remove,
  .open                = ftdi_open,
  .close               = ftdi_close,
  .throttle            = ftdi_throttle,
  .unthrottle          = ftdi_unthrottle,
  .write               = ftdi_write,
  .write_room          = ftdi_write_room,
  .chars_in_buffer     = ftdi_chars_in_buffer,
  .read_bulk_callback  = ftdi_read_bulk_callback,
  .write_bulk_callback = ftdi_write_bulk_callback,
  /* ... */
};

/* Driver Initialization */
static int __init ftdi_init(void)
{
   /* ... */
   /* Register with the USB-Serial core */
   retval = usb_serial_register(&ftdi_sio_device);
   /* ... */
   /* Register with the USB core */
   retval = usb_register(&ftdi_driver);
   /* ... */
}
```

Human Interface Devices

Devices such as keyboards and mice are called *human interface devices* (HIDs). Take a look at the section "USB and Bluetooth Keyboards" in Chapter 7, "Input Drivers," for a discussion on the USB HID class.

Bluetooth

A USB-Bluetooth dongle is a quick way to Bluetooth-enable your computer so that it can communicate with Bluetooth-equipped devices such as cell phones, mice, or handhelds. Chapter 16 discusses the USB Bluetooth class.

Gadget Drivers

In a typical usage scenario, an embedded device connects to a PC host over USB. Embedded computers usually belong to the device side of USB, unlike PC systems that function as USB hosts. Because Linux runs on both embedded and PC systems, it needs support to run on either end of USB. The USB Gadget project brings USB device mode capability to embedded Linux systems. Bus 3 of the embedded Linux device in Figure 11.2 can, for example, use a *gadget driver* to let the device function as a mass storage drive when connected to a host computer.

Before proceeding, let's briefly look at some related terminology. The USB controller at the device side is variously called a *device controller*, *peripheral controller*, *client controller*, or *function controller*. The terms *gadget* and *gadget driver* are commonly used rather than the heavily overloaded words *device* and *device driver*.

USB gadget support is now part of the mainline kernel and contains the following:

- Drivers for USB device controllers integrated into SoC families such as Intel PXA, Texas Instruments OMAP, and Atmel AT91. These drivers additionally provide a *gadget API* that gadget drivers can use.

- Gadget drivers for device classes such as storage, networking, and serial converters. These drivers answer to their class when they receive enumeration requests from host-side software. A storage gadget driver, for example, identifies itself as a class `0x08` (mass storage class) device and exports a storage partition to the host. You can specify the associated block device node or filename via a module-insertion parameter. Because the exported region has to appear to the host as a mass storage device, the gadget driver implements the SCSI interactions required by the USB mass storage protocol. Gadget drivers are also available for Ethernet and serial devices.

- A skeletal gadget driver, *drivers/usb/gadget/zero.c*, that you may use to test device controller drivers.

Gadget drivers use the services of the gadget API provided by device controller drivers. They populate a `usb_gadget_driver` structure and register it with the kernel using `usb_gadget_register_driver()`. Hardware specifics are hidden inside the gadget API implementation offered by individual device controller drivers, so the gadget drivers themselves are hardware independent.

Documentation/DocBook/gadget.tmpl provides an overview of the gadget API. Have a look at http://linux-usb.org/gadget/ for more on the gadget project.

Debugging

A USB bus analyzer magnifies the goings-on in the bus and is useful for debugging low-level problems. If you can't get hold of an analyzer, you might be able to make do with the kernel's soft USB tracer, *usbmon.* This tool captures traffic between USB host controllers and devices. To collect a trace, read from the *debugfs*[3] file */sys/kernel/debug/usbmon/Xt*, where *X* is the bus number to which your device is connected.

[3] An in-memory filesystem to export kernel debug data to user space.

For example, consider a USB disk connected to a PC. From the associated "T:" line in */proc/bus/usb/devices*, you can see that the drive is attached to bus 1:

```
T: Bus=01 Lev=01 Prnt=01 Port=03 Cnt=01 Dev#= 2 Spd=480 MxCh= 0
```

Ensure that you have enabled *debugfs* (`CONFIG_DEBUG_FS`) and usbmon (`CONFIG_USB_MON`) support in your kernel. This is a snapshot of usbmon output while copying a file from the disk:

Code View:

```
bash> mount -t debugfs none_debugs /sys/kernel/debug/
bash> cat /sys/kernel/debug/usbmon/1u
...
ee6a5c40 3718782540 S Bi:1:002:1 -115 20480 <
ee6a5cc0 3718782567 S Bi:1:002:1 -115 65536 <
ee6a5d40 3718782595 S Bi:1:002:1 -115 36864 <
ee6a5c40 3718788189 C Bi:1:002:1 0 20480 = 0f846801 118498f\ 15c60500 01680106
5e846801 608498fe 6f280087 68000000
ee6a5cc0 3718800994 C Bi:1:002:1 0 65536 = 118498fe 15c60500\ 01680106 5e846801
608498fe 6f280087 68000000 00884800
ee6a5d40 3718801001 C Bi:1:002:1 0 36864 = 13608498 fe4f4a01\ 00514a01 006f2800
87680000 00008848 00000100 b7f00100
...
```

Each output line starts with the URB address, followed by an event timestamp. An *S* in the next column indicates URB submission, and a *C* announces a callback. The following field has the format `URBType:Bus#:DeviceAddress:Endpoint#`. In the preceding output, a `URBType` of `Bi` stands for a bulk URB in the `IN` direction. After this, usbmon dumps the URB status, data length, a data tag (`=` or `<` in the preceding output), and the data words (if the tag is `=`). The last three lines in the preceding output are callbacks associated with bulk URBs submitted in earlier lines. You can match the callbacks with the related submissions using the URB addresses. *Documentation/usb/usbmon.txt* details usbmon syntax and contains example code to parse the output into human readable form.

If you turn on *Device Drivers* ⟶ *USB Support* ⟶ *USB Verbose Debug Messages* during kernel configuration, the kernel will emit the contents of all `dev_dbg()` statements present in the USB subsystem.

You can glean device and bus specific information from the USB filesystem (*usbfs*) node, */proc/bus/usb/devices*. And as we discuss in Chapter 19, "Drivers in User Space," usbfs also lets you implement USB device drivers in user space. Even when the final destination of your USB driver is inside the kernel, starting with a user-space driver can ease debugging and testing.

The linux-usb-devel mailing list is the forum to discuss questions related to USB device drivers. Visit https://lists.sourceforge.net/lists/listinfo/linux-usb-devel for subscription and archive retrieval information. Read www.linux-usb.org/usbtest for ideas on USB testing.

The home page of the Linux-USB project is www.linux-usb.org. You may download the USB 2.0 specification, OTG supplement, and other related standards from www.usb.org/developers/docs.

Looking at the Sources

The USB core layer lives in *drivers/usb/core/.* This directory also contains URB manipulation routines and the usbfs implementation. The hub driver and khubd are part of *drivers/usb/core/hub.c.* The *drivers/usb/host/* directory contains host controller device drivers. USB-related header definitions reside in *include/linux/usb*.h.* The usbmon tracer is in *drivers/usb/mon/.* Look inside *Documentation/usb/* for Linux-USB documentation.

USB class drivers stay in various subdirectories under *drivers/usb/.* The mass storage driver *drivers/usb/storage/*, in tandem with the SCSI subsystem *drivers/scsi/*, implements the USB mass storage protocol. The *drivers/input/*[4] directory tree includes drivers for USB input devices such as keyboards and mice; *drivers/usb/serial/* has drivers for USB-to-serial converters; *drivers/usb/media/* supports USB multimedia devices; *drivers/net/usb/*[5] has drivers for USB Ethernet dongles; and *drivers/usb/misc/* contains drivers for miscellaneous USB devices such as LEDs, LCDs, and fingerprint sensors. Look at *drivers/usb/usb-skeleton.c* for a starting point driver template if you can't zero in on a closer match.

[4] Before the 2.6.22 kernel release, USB input device drivers used to reside in *drivers/usb/input/*.

[5] Before the 2.6.22 kernel release, USB network device drivers used to reside in *drivers/usb/net/*.

The USB gadget subsystem is in *drivers/usb/gadget/.* This directory contains USB device controller drivers, and gadget drivers for mass storage (*file_storage.c*), serial converters (*serial.c*), and Ethernet networking (*ether.c*).

Table 11.3 contains the main data structures used in this chapter and their location in the source tree. Table 11.4 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 11.3. Summary of Data Structures

Data Structure	Location	Description
`urb`	*include/linux/usb.h*	Centerpiece of the USB data transfer mechanism
`pipe`	*include/linux/usb.h*	Address element of a URB

`usb_device_descriptor`
`usb_config_descriptor`
`usb_interface_descriptor`
`usb_endpoint_descriptor`
	include/linux/usb/ch9.h	Descriptors that hold information about a USB device

`usb_device`	*include/linux/usb.h*	Representation of a USB device
`usb_device_id`	*include/linux/mod_devicetable.h*	Identity of a USB device
`usb_driver`	*include/linux/usb.h*	Representation of a USB client driver
`usb_gadget_driver`	*include/linux/usb_gadget.h*	Representation of a USB gadget driver

Table 11.4. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`usb_register()`	*include/linux/usb.h* *drivers/usb/core/driver.c*	Registers a `usb_driver` with the USB core
`usb_deregister()`	*drivers/usb/core/driver.c*	Unregisters a `usb_driver` from the USB core
`usb_set_intfdata()`	*include/linux/usb.h*	Attaches device-specific data to a `usb_interface`
`usb_get_intfdata()`	*include/linux/usb.h*	Detaches device-specific data from a `usb_interface`
`usb_register_dev()`	*drivers/usb/core/file.c*	Associates a character interface with a USB client driver
`usb_deregister_dev()`	*drivers/usb/core/file.c*	Dissociates a character interface from a USB client driver
`usb_alloc_urb()`	*drivers/usb/core/urb.c*	Allocates a URB
`usb_fill_[control|int|bulk]_urb()`	*include/linux/usb.h*	Populates a URB
`usb_[control|interrupt|bulk]_msg()`	*drivers/usb/core/message.c*	Wrappers for synchronous URB submission
`usb_submit_urb()`	*drivers/usb/core/urb.c*	Submits a URB to the USB core
`usb_free_urb()`	*drivers/usb/core/urb.c*	Frees references to a completed URB
`usb_unlink_urb()`	*drivers/usb/core/urb.c*	Frees references to a pending URB
`usb_[rcv|snd][ctrl|int|bulk|isoc]pipe()`	*include/linux/usb.h*	Creates a USB pipe
`usb_find_interface()`	*drivers/usb/core/usb.c*	Gets the `usb_interface` associated with a USB client driver
`usb_buffer_alloc()`	*drivers/usb/core/usb.c*	Allocates a consistent DMA transfer buffer
`usb_buffer_free()`	*drivers/usb/core/usb.c*	Frees a buffer that was allocated using `usb_buffer_alloc()`
`usb_serial_register()`	*drivers/usb/serial/usb-serial.c*	Registers a driver with the USB-Serial core
`usb_serial_deregister()`	*drivers/usb/serial/usb-serial.c*	Unregisters a driver from the USB-Serial core
`usb_gadget_register_driver()`	Device controller drivers in *drivers/usb/gadget/*	Registers a gadget with a device controller driver

Chapter 12. Video Drivers

In This Chapter

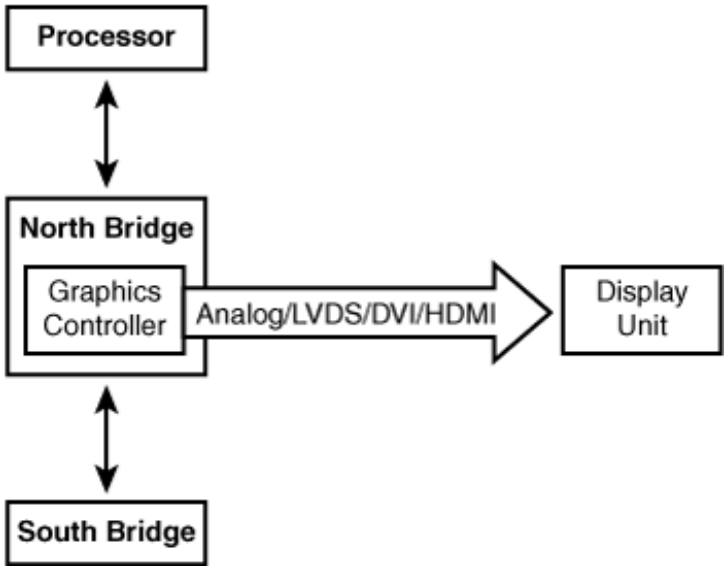
- Display Architecture 356
- Linux-Video Subsystem 359
- Display Parameters 361
- The Frame Buffer API 362
- Frame Buffer Drivers 365
- Console Drivers 380
- Debugging 387
- Looking at the Sources 388

Video hardware generates visual output for a computer system to display. In this chapter, let's find out how the kernel supports video controllers and discover the advantages offered by the frame buffer abstraction. Let's also learn to write console drivers that display messages emitted by the kernel.

Display Architecture

Figure 12.1 shows the display assembly on a PC-compatible system. The graphics controller that is part of the North Bridge (see the sidebar "The North Bridge") connects to different types of display devices using several interface standards (see the sidebar "Video Cabling Standards").

Figure 12.1. Display connection on a PC system.



Video Graphics Array (VGA) is the original display standard introduced by IBM, but it's more of a resolution specification today. VGA refers to a resolution of 640x480, whereas newer standards such as *eXtended Graphics Array* (XGA) and *Super Video Graphics Array* (SVGA) support higher resolutions of 800x600 and 1024x768. *Quarter VGA* (QVGA) panels having a resolution of 320x240 are common on embedded devices, such as handhelds and smart phones.

Graphics controllers in the x86 world compatible with VGA and its derivatives offer a character-based text mode and a pixel-based graphics mode. The non-x86 embedded space is non-VGA, however, and has no concept of a dedicated text mode.

The North Bridge

In earlier chapters, you learned about peripheral buses such as LPC, I^2C, PCMCIA, PCI, and USB, all of which are sourced from the South Bridge on PC-centric systems. Display architecture, however, takes us inside the North Bridge. A North Bridge in the Intel-based PC architecture is either a *Graphics and Memory Controller Hub* (GMCH) or a *Memory Controller Hub* (MCH). The former contains a memory controller, a *Front Side Bus* (FSB) controller, and a graphics controller. The latter lacks an integrated graphics controller but provides an *Accelerated Graphics Port* (AGP) channel to connect external graphics hardware.

Consider, for example, the Intel 855 GMCH North Bridge chipset. The FSB controller in the 855 GMCH interfaces with Pentium M processors. The memory controller supports *Dual Data Rate* (DDR) SDRAM memory chips. The integrated graphics controller lets you connect to display devices using analog VGA, LVDS, or DVI (see the sidebar "Video Cabling Standards"). The 855 GMCH enables you to simultaneously send output to two displays, so you can, for example, dispatch similar or separate information to your laptop's LCD panel and an external CRT monitor at the same time.

Recent North Bridge chipsets, such as the AMD 690G, include support for HDMI (see the following sidebar) in addition to VGA and DVI.

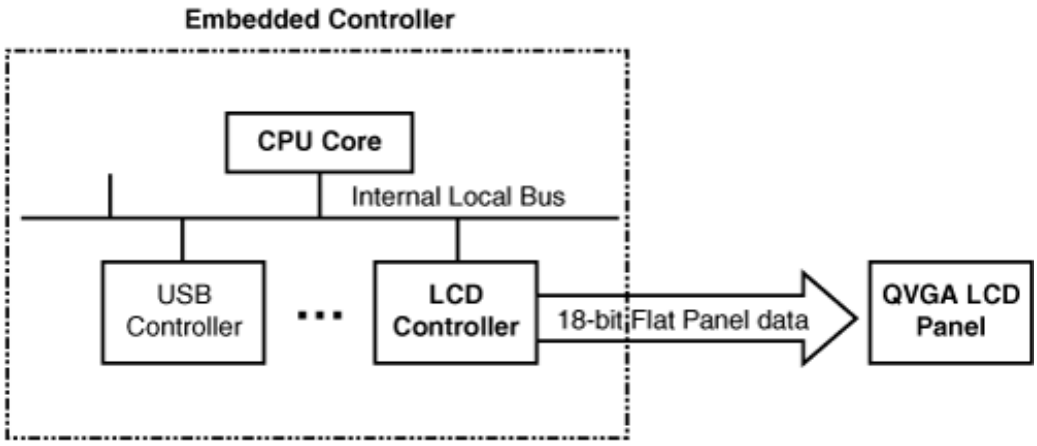
Video Cabling Standards

Several interfacing standards specify the connection between video controllers and display devices. Display devices and the cabling technologies they use follow:

- An analog display such as a *cathode ray tube* (CRT) monitor that has a standard VGA connector.
- A digital flat-panel display such as a laptop *Thin Film Transistor* (TFT) LCD that has a *low-voltage differential signaling* (LVDS) connector.
- A display monitor that complies with the *Digital Visual Interface* (DVI) specification. DVI is a standard developed by the *Digital Display Working Group* (DDWG) for carrying high-quality video. There are three DVI subclasses: *digital-only* (DVI-D), *analog-only* (DVI-A), and *digital-and-analog* (DVI-I).
- A display monitor that complies with the *High-Definition Television* (HDTV) specification using the *High-Definition Multimedia Interface* (HDMI). HDMI is a modern digital audio-video cable standard that supports high data rates. Unlike video-only standards such as DVI, HDMI can carry both picture and sound.

Embedded SoCs usually have an on-chip LCD controller, as shown in Figure 12.2. The output emanating from the LCD controller are TTL (*Transistor-Transistor Logic*) signals that pack 18 bits of flat-panel video data, six each for the three primary colors, red, green, and blue. Several handhelds and phones use QVGA-type internal LCD panels that directly receive the TTL flat-panel video data sourced by LCD controllers.

Figure 12.2. Display connection on an embedded system.

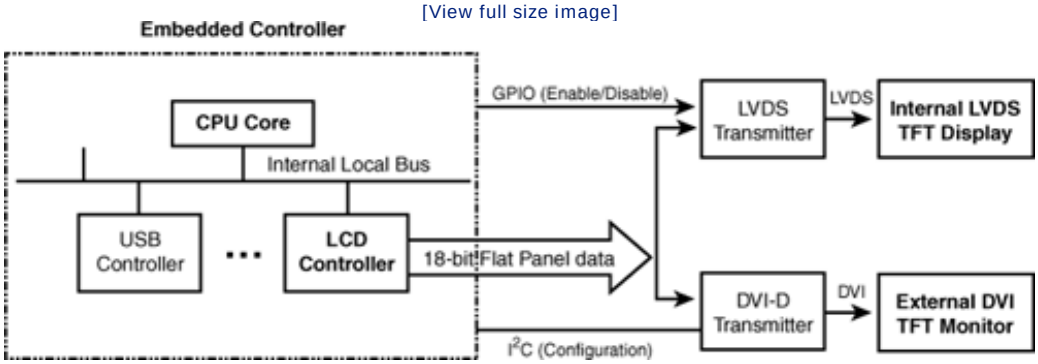


The embedded device, as in Figure 12.3, supports dual display panels: an internal LVDS flat-panel LCD and an

external DVI monitor. The internal TFT LCD takes an LVDS connector as input, so an LVDS transmitter chip is used to convert the flat-panel signals to LVDS. An example of an LVDS transmitter chip is DS90C363B from National Semiconductor. The external DVI monitor takes only a DVI connector, so a DVI transmitter is used to convert the 18-bit video signals to DVI-D. An I^2C interface is provided so that the device driver can configure the DVI transmitter registers. An example of a DVI transmitter chip is SiI164 from Silicon Image.

Figure 12.3. LVDS and DVI connections on an embedded system.

[View full size image]



Chapter 12. Video Drivers

In This Chapter

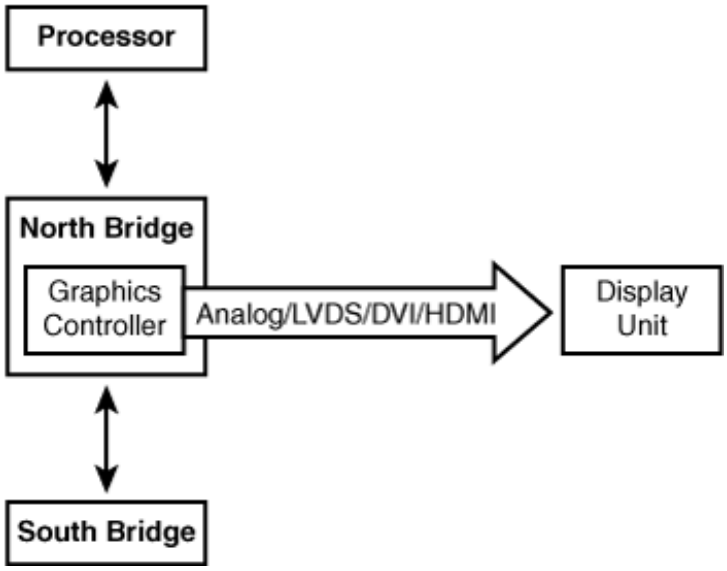
- Display Architecture 356
- Linux-Video Subsystem 359
- Display Parameters 361
- The Frame Buffer API 362
- Frame Buffer Drivers 365
- Console Drivers 380
- Debugging 387
- Looking at the Sources 388

Video hardware generates visual output for a computer system to display. In this chapter, let's find out how the kernel supports video controllers and discover the advantages offered by the frame buffer abstraction. Let's also learn to write console drivers that display messages emitted by the kernel.

Display Architecture

Figure 12.1 shows the display assembly on a PC-compatible system. The graphics controller that is part of the North Bridge (see the sidebar "The North Bridge") connects to different types of display devices using several interface standards (see the sidebar "Video Cabling Standards").

Figure 12.1. Display connection on a PC system.



Video Graphics Array (VGA) is the original display standard introduced by IBM, but it's more of a resolution specification today. VGA refers to a resolution of 640x480, whereas newer standards such as *eXtended Graphics Array* (XGA) and *Super Video Graphics Array* (SVGA) support higher resolutions of 800x600 and 1024x768. *Quarter VGA* (QVGA) panels having a resolution of 320x240 are common on embedded devices, such as handhelds and smart phones.

Graphics controllers in the x86 world compatible with VGA and its derivatives offer a character-based text mode and a pixel-based graphics mode. The non-x86 embedded space is non-VGA, however, and has no concept of a dedicated text mode.

The North Bridge

In earlier chapters, you learned about peripheral buses such as LPC, I^2C, PCMCIA, PCI, and USB, all of which are sourced from the South Bridge on PC-centric systems. Display architecture, however, takes us inside the North Bridge. A North Bridge in the Intel-based PC architecture is either a *Graphics and Memory Controller Hub* (GMCH) or a *Memory Controller Hub* (MCH). The former contains a memory controller, a *Front Side Bus* (FSB) controller, and a graphics controller. The latter lacks an integrated graphics controller but provides an *Accelerated Graphics Port* (AGP) channel to connect external graphics hardware.

Consider, for example, the Intel 855 GMCH North Bridge chipset. The FSB controller in the 855 GMCH interfaces with Pentium M processors. The memory controller supports *Dual Data Rate* (DDR) SDRAM memory chips. The integrated graphics controller lets you connect to display devices using analog VGA, LVDS, or DVI (see the sidebar "Video Cabling Standards"). The 855 GMCH enables you to simultaneously send output to two displays, so you can, for example, dispatch similar or separate information to your laptop's LCD panel and an external CRT monitor at the same time.

Recent North Bridge chipsets, such as the AMD 690G, include support for HDMI (see the following sidebar) in addition to VGA and DVI.

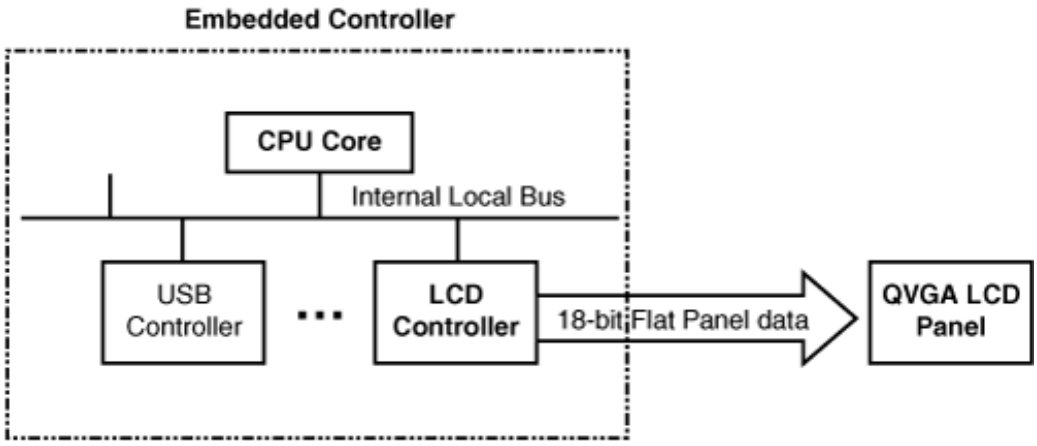
Video Cabling Standards

Several interfacing standards specify the connection between video controllers and display devices. Display devices and the cabling technologies they use follow:

- An analog display such as a *cathode ray tube* (CRT) monitor that has a standard VGA connector.
- A digital flat-panel display such as a laptop *Thin Film Transistor* (TFT) LCD that has a *low-voltage differential signaling* (LVDS) connector.
- A display monitor that complies with the *Digital Visual Interface* (DVI) specification. DVI is a standard developed by the *Digital Display Working Group* (DDWG) for carrying high-quality video. There are three DVI subclasses: *digital-only* (DVI-D), *analog-only* (DVI-A), and *digital-and-analog* (DVI-I).
- A display monitor that complies with the *High-Definition Television* (HDTV) specification using the *High-Definition Multimedia Interface* (HDMI). HDMI is a modern digital audio-video cable standard that supports high data rates. Unlike video-only standards such as DVI, HDMI can carry both picture and sound.

Embedded SoCs usually have an on-chip LCD controller, as shown in Figure 12.2. The output emanating from the LCD controller are TTL (*Transistor-Transistor Logic*) signals that pack 18 bits of flat-panel video data, six each for the three primary colors, red, green, and blue. Several handhelds and phones use QVGA-type internal LCD panels that directly receive the TTL flat-panel video data sourced by LCD controllers.

Figure 12.2. Display connection on an embedded system.

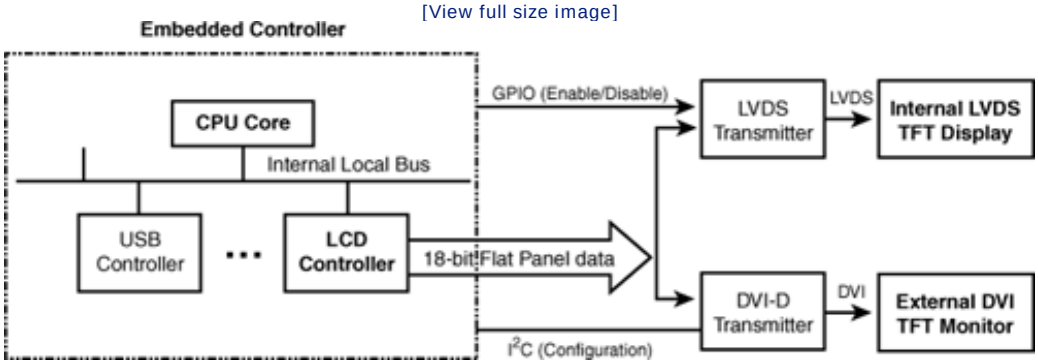


The embedded device, as in Figure 12.3, supports dual display panels: an internal LVDS flat-panel LCD and an

external DVI monitor. The internal TFT LCD takes an LVDS connector as input, so an LVDS transmitter chip is used to convert the flat-panel signals to LVDS. An example of an LVDS transmitter chip is DS90C363B from National Semiconductor. The external DVI monitor takes only a DVI connector, so a DVI transmitter is used to convert the 18-bit video signals to DVI-D. An I^2C interface is provided so that the device driver can configure the DVI transmitter registers. An example of a DVI transmitter chip is SiI164 from Silicon Image.

Figure 12.3. LVDS and DVI connections on an embedded system.

[View full size image]

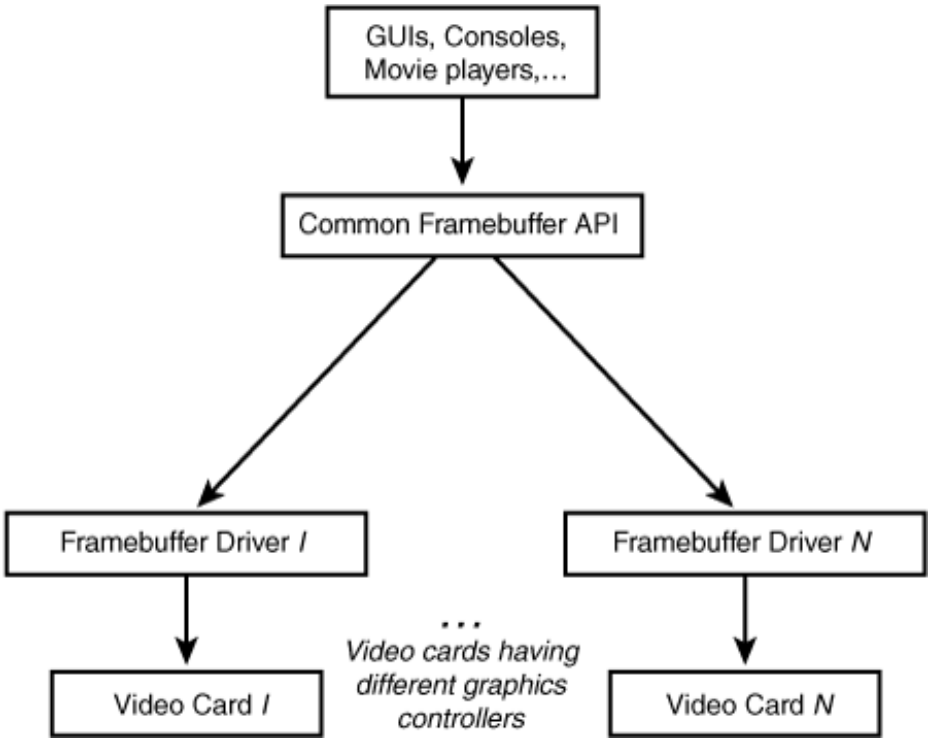


Linux-Video Subsystem

The concept of frame buffers is central to video on Linux, so let's first find out what that offers.

Because video adapters can be based on different hardware architectures, the implementation of higher kernel layers and applications might need to vary across video cards. This results in nonuniform schemes to handle different video cards. The ensuing nonportability and extra code necessitate greater investment and maintenance. The frame buffer concept solves this problem by describing a general abstraction and specifying a programming interface that allows applications and higher kernel layers to be written in a platform-independent manner. Figure 12.4 shows you the frame buffer advantage.

Figure 12.4. The frame buffer advantage.



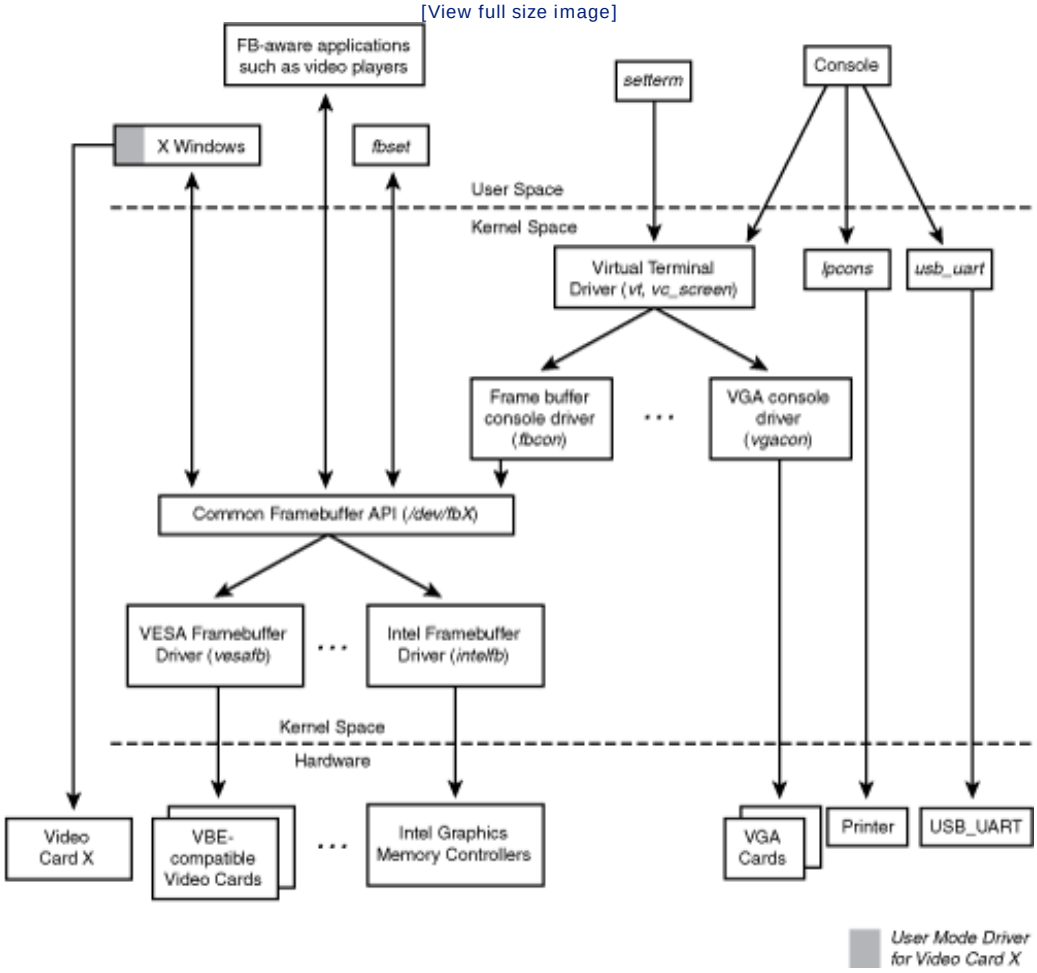
The kernel's frame buffer interface thus allows applications to be independent of the vagaries of the underlying graphics hardware. Applications run unchanged over diverse types of video hardware if they and the display drivers conform to the frame buffer interface. As you will soon find out, the common frame buffer programming interface also brings hardware independence to kernel layers, such as the frame buffer console driver.

Today, several applications, such as web browsers and movie players, work directly over the frame buffer interface. Such applications can do graphics without help from a windowing system.

The X Windows server (Xfbdev) is capable of working over the frame buffer interface, as shown in Figure 12.5.

Figure 12.5. Linux-Video subsystem.

[View full size image]



The Linux-Video subsystem shown in Figure 12.5 is a collection of low-level display drivers, middle-level frame buffer and console layers, a high-level virtual terminal driver, user mode drivers part of X Windows, and utilities to configure display parameters. Let's trace the figure top down:

- The X Windows GUI has two options for operating over video cards. It can use either a suitable built-in user-space driver for the card or work over the frame buffer subsystem.

- Text mode consoles function over the virtual terminal character driver. Virtual terminals, introduced in the section "TTY Drivers" in Chapter 6, "Serial Drivers," are full-screen text-based terminals that you get when you logon in text mode. Like X Windows, text consoles have two operational choices. They can either work over a card-specific console driver, or use the generic frame buffer console driver (*fbcon*) if the kernel supports a low-level frame buffer driver for the card in question.

Display Parameters

Sometimes, configuring the properties associated with your display panel might be the only driver changes that you need to make to enable video on your device, so let's start learning about video drivers by looking at common display parameters. We will assume that the associated driver conforms to the frame buffer interface, and use the *fbset* utility to obtain display characteristics:

```
bash> fbset
mode "1024x768-60"
    # D: 65.003 MHz, H: 48.365 kHz, V: 60.006 Hz
    geometry 1024 768 1024 768 8
    timings 15384 168 16 30 2 136 6
    hsync high
    vsync high
    rgba 8/0,8/0,8/0,0/0
endmode
```

The `D:` value in the output stands for the *dotclock*, which is the speed at which the video hardware draws pixels on the display. The value of 65.003MHz in the preceding output means that it'll take (1/65.003*1000000) or about 15,384 picoseconds for the video controller to draw a single pixel. This duration is called the *pixclock* and is shown as the first numeric parameter in the line starting with `timings`. The numbers against "geometry" announce that the visible and virtual resolutions are 1024x768 (SVGA) and that the bits required to store information pertaining to a pixel is 8.

The `H:` value specifies the horizontal scan rate, which is the number of horizontal display lines scanned by the video hardware in one second. This is the inverse of the pixclock times the X-resolution. The `V:` value is the rate at which the entire display is refreshed. This is the inverse of the pixclock times the visible X-resolution times the visible Y-resolution, which is around 60Hz in this example. In other words, the LCD is refreshed 60 times in a second.

Video controllers issue a horizontal sync (`HSYNC`) pulse at the end of each line and a vertical sync (`VSYNC`) pulse after each display frame. The durations of `HSYNC` (in terms of pixels) and `VSYNC` (in terms of pixel lines) are shown as the last two parameters in the line starting with "`timings`." The larger your display, the bigger the likely values of `HSYNC` and `VSYNC`. The four numbers before the `HSYNC` duration in the `timings` line announce the length of the right display margin (or horizontal front porch), left margin (or horizontal back porch), lower margin (or vertical front porch), and upper margin (or vertical back porch), respectively. *Documentation/fb/framebuffer.txt* and the man page of *fb.modes* pictorially show these parameters.

To tie these parameters together, let's calculate the pixclock value for a given refresh rate, which is 60.006Hz in our example:

```
dotclock = (X-resolution + left margin + right margin
           + HSYNC length) * (Y-resolution + upper margin
           + lower margin + VSYNC length) * refresh rate
         = (1024 + 168 + 16 + 136) * (768 + 30 + 2 + 6) * 60.006
         = 65.003 MHz
pixclock = 1/dotclock
         = 15384 picoseconds (which matches with the fbset output
                              above)
```

The Frame Buffer API

Let's next wet our feet in the frame buffer API. The frame buffer core layer exports device nodes to user space so that applications can access each supported video device. */dev/fbX* is the node associated with frame buffer device *X.* The following are the main data structures that interest users of the frame buffer API. Inside the kernel, they are defined in *include/linux/fb.h*, whereas in user land, their definitions reside in */usr/include/linux/fb.h*:

1. Variable information pertaining to the video card that you saw in the fbset output in the previous section is held in `struct fb_var_screeninfo`. This structure contains fields such as the X-resolution, Y-resolution, bits required to hold a pixel, pixclock, `HSYNC` duration, `VSYNC` duration, and margin lengths. These values are programmable by the user:

```
struct fb_var_screeninfo {
  __u32 xres;             /* Visible resolution in the X axis */
  __u32 yres;             /* Visible resolution in the Y axis */
  /* ... */
  __u32 bits_per_pixel;   /* Number of bits required to hold a
                             pixel */
  /* ... */
  __u32 pixclock;         /* Pixel clock in picoseconds */
  __u32 left_margin;      /* Time from sync to picture */
  __u32 right_margin;     /* Time from picture to sync */
  /* ... */
  __u32 hsync_len;        /* Length of horizontal sync */
  __u32 vsync_len;        /* Length of vertical sync */
  /* ... */
};
```

2. Fixed information about the video hardware, such as the start address and size of frame buffer memory, is held in `struct fb_fix_screeninfo`. These values cannot be altered by the user:

```
struct fb_fix_screeninfo {
  char id[16];              /* Identification string */
  unsigned long smem_start; /* Start of frame buffer memory */
  __u32 smem_len;           /* Length of frame buffer memory */
  /* ... */
};
```

3. The `fb_cmap` structure specifies the color map, which is used to convey the user's definition of colors to the underlying video hardware. You can use this structure to define the RGB (Red, Green, Blue) ratio that you desire for different colors:

```
struct fb_cmap {
  __u32 start;    /* First entry */
  __u32 len;      /* Number of entries */
  __u16 *red;     /* Red values */
  __u16 *green;   /* Green values */
  __u16 *blue;    /* Blue values */
  __u16 *transp;  /* Transparency. Discussed later on */
};
```

Listing 12.1 is a simple application that works over the frame buffer API. The program clears the screen by operating on */dev/fb0*, the frame buffer device node corresponding to the display. It first deciphers the visible resolutions and the *bits per pixel* in a hardware-independent manner using the frame buffer API, `FBIOGET_VSCREENINFO`. This interface command gleans the display's variable parameters by operating on the `fb_var_screeninfo` structure. The program then goes on to `mmap()` the frame buffer memory and clears each constituent pixel bit.

Listing 12.1. Clear the Display in a Hardware-Independent Manner

Code View:

```
#include <stdio.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>
#include <stdlib.h>

struct fb_var_screeninfo vinfo;

int
main(int argc, char *argv[])
{
  int fbfd, fbsize, i;
  unsigned char *fbbuf;

  /* Open video memory */
  if ((fbfd = open("/dev/fb0", O_RDWR)) < 0) {
    exit(1);
  }

  /* Get variable display parameters */
  if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
    printf("Bad vscreeninfo ioctl\n");
    exit(2);
  }

  /* Size of frame buffer =
         (X-resolution * Y-resolution * bytes per pixel) */
  fbsize = vinfo.xres*vinfo.yres*(vinfo.bits_per_pixel/8);

  /* Map video memory */
  if ((fbbuf = mmap(0, fbsize, PROT_READ|PROT_WRITE,
                       MAP_SHARED, fbfd, 0)) == (void *) -1){
    exit(3);
  }

  /* Clear the screen */
   for (i=0; i<fbsize; i++) {
     *(fbbuf+i) = 0x0;
   }

   munmap(fbbuf, fbsize);
   close(fbfd);
}
```

We look at another frame buffer application when we learn to access memory regions from user space in Chapter 19, "Drivers in User Space."

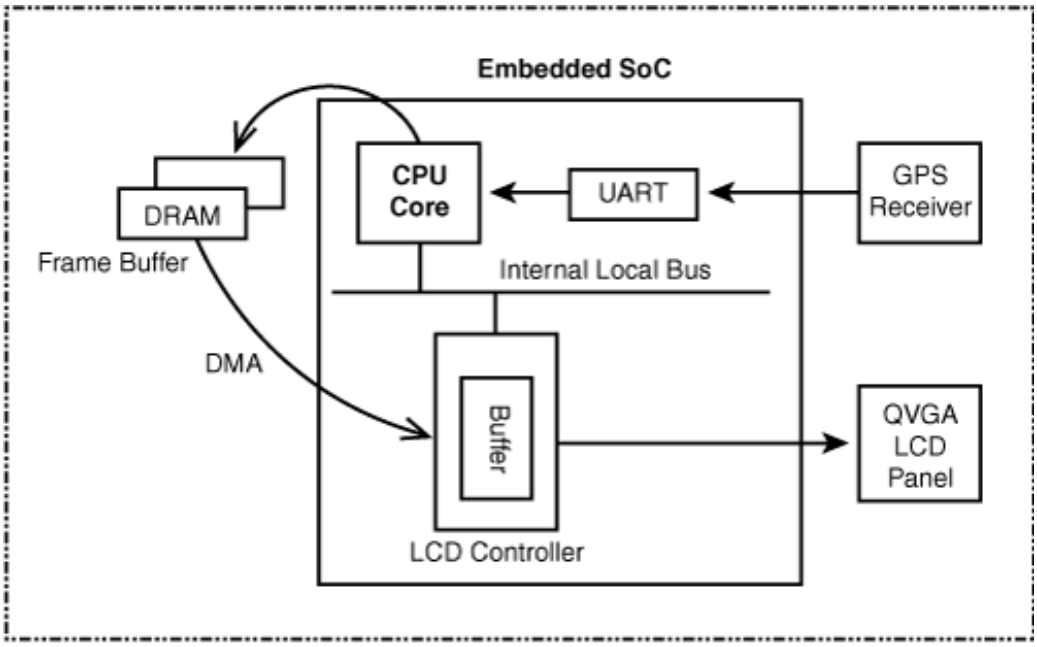
Frame Buffer Drivers

Now that you have an idea of the frame buffer API and how it provides hardware independence, let's discover the architecture of a low-level frame buffer device driver using the example of a navigation system.

Device Example: Navigation System

Figure 12.6 shows video operation on an example vehicle navigation system built around an embedded SoC. A GPS receiver streams coordinates to the SoC via a UART interface. An application produces graphics from the received location information and updates a frame buffer in system memory. The frame buffer driver DMAs this picture data to display buffers that are part of the SoC's LCD controller. The controller forwards the pixel data to the QVGA LCD panel for display.

Figure 12.6. Display on a Linux navigation device.



Our goal is to develop the video software for this system. Let's assume that Linux supports the SoC used on this navigation device and that all architecture-dependent interfaces such as DMA are supported by the kernel.

One possible hardware implementation of the device shown in Figure 12.6 is by using a Freescale i.MX21 SoC. The CPU core in that case is an ARM9 core, and the on-chip video controller is the Liquid Crystal Display Controller (LCDC). SoCs commonly have a high-performance internal local bus that connects to controllers such as DRAM and video. In the case of the iMX.21, this bus is called the Advanced High-Performance Bus (AHB). The LCDC connects to the AHB.

The navigation system's video software is broadly architected as a GPS application operating over a low-level frame buffer driver for the LCD controller. The application fetches location coordinates from the GPS receiver by reading */dev/ttySX*, where *X* is the UART number connected to the receiver. It then translates the geographic fix information into a picture and writes the pixel data to the frame buffer associated with the LCD controller. This is done on the lines of Listing 12.1, except that picture data is dispatched rather than zeros to clear the screen.

The rest of this section focuses only on the low-level frame buffer device driver. Like many other driver subsystems, the full complement of facilities, modes, and options offered by the frame buffer core layer are complex and can be learned only with coding experience. The frame buffer driver for the example navigation system is relatively simplistic and is only a starting point for deeper explorations.

Table 12.1 describes the register model of the LCD controller shown in Figure 12.6. The frame buffer driver in Listing 12.2 operates over these registers.

Table 12.1. Register Layout of the LCD Controller Shown in Figure 12.6

Register Name	Used to Configure
`SIZE_REG`	LCD panel's maximum X and Y dimensions
`HSYNC_REG`	`HSYNC` duration
`VSYNC_REG`	`VSYNC` duration
`CONF_REG`	Bits per pixel, pixel polarity, clock dividers for generating pixclock, color/monochrome mode, and so on
`CTRL_REG`	Enable/disable LCD controller, clocks, and DMA
`DMA_REG`	Frame buffer's DMA start address, burst length, and watermark sizes
`STATUS_REG`	Status values
`CONTRAST_REG`	Contrast level

Our frame buffer driver (called *myfb*) is implemented as a platform driver in Listing 12.2. As you learned in Chapter 6, a platform is a pseudo bus usually used to connect lightweight devices integrated into SoCs, with the kernel's device model. Architecture-specific setup code (in *arch/your-arch/your-platform/*) adds the platform using `platform_device_add()`; but for simplicity, the `probe()` method of the myfb driver performs this before registering itself as a platform driver. Refer back to the section "Device Example: Cell Phone" in Chapter 6 for the general architecture of a platform driver and associated entry points.

Data Structures

Let's take a look at the major data structures and methods associated with frame buffer drivers and then zoom in on myfb. The following two are the main structures:

1. `struct fb_info` is the centerpiece data structure of frame buffer drivers. This structure is defined in *include/linux/fb.h* as follows:

```
struct fb_info {
  /* ... */
  struct fb_var_screeninfo var;    /* Variable screen information.
```

```
                                        Discussed earlier. */
  struct fb_fix_screeninfo fix;    /* Fixed screen information.
                                        Discussed earlier. */
  /* ... */
  struct fb_cmap cmap;             /* Color map.
                                        Discussed earlier. */
  /* ... */
  struct fb_ops *fbops;            /* Driver operations.
                                        Discussed next. */
  /* ... */
  char __iomem *screen_base;       /* Frame buffer's
                                        virtual address */
  unsigned long screen_size;       /* Frame buffer's size */
  /* ... */
  /* From here on everything is device dependent */
  void *par;                       /* Private area */
};
```

Memory for `fb_info` is allocated by `framebuffer_alloc()`, a library routine provided by the frame buffer core. This function also takes the size of a private area as an argument and appends that to the end of the allocated `fb_info`. This private area can be referenced using the `par` pointer in the `fb_info` structure. The semantics of `fb_info` fields such as `fb_var_screeninfo` and `fb_fix_screeninfo` were discussed in the section "The Frame Buffer API."

2. The `fb_ops` structure contains the addresses of all entry points provided by the low-level frame buffer driver. The first few methods in `fb_ops` are necessary for the functioning of the driver, while the remaining are optional ones that provide for graphics acceleration. The responsibility of each function is briefly explained within comments:

   Code View:

```
struct fb_ops {
  struct module *owner;
  /* Driver open */
  int (*fb_open)(struct fb_info *info, int user);
  /* Driver close */
  int (*fb_release)(struct fb_info *info, int user);
  /* ... */
  /* Sanity check on video parameters */
  int (*fb_check_var)(struct fb_var_screeninfo *var,
                      struct fb_info *info);
  /* Configure the video controller registers */
  int (*fb_set_par)(struct fb_info *info);
  /* Create pseudo color palette map */
  int (*fb_setcolreg)(unsigned regno, unsigned red,
                unsigned green, unsigned blue,
                unsigned transp, struct fb_info *info);
  /* Blank/unblank display */
  int (*fb_blank)(int blank, struct fb_info *info);
  /* ... */
  /* Accelerated method to fill a rectangle with pixel lines */
  void (*fb_fillrect)(struct fb_info *info,
                      const struct fb_fillrect *rect);
  /* Accelerated method to copy a rectangular area from one
     screen region to another */
  void (*fb_copyarea)(struct fb_info *info,
```

```
                    const struct fb_copyarea *region);
  /* Accelerated method to draw an image to the display */
  void (*fb_imageblit)(struct fb_info *info,
                       const struct fb_image *image);
  /* Accelerated method to rotate the display */
  void (*fb_rotate)(struct fb_info *info, int angle);
  /* Ioctl interface to support device-specific commands */
  int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,
                  unsigned long arg);
  /* ... */
};
```

Let's now look at the driver methods that Listing 12.2 implements for the myfb driver.

Checking and Setting Parameters

The `fb_check_var()` method performs a sanity check of variables such as X-resolution, Y-resolution, and bits per pixel. So, if you use fbset to set an X-resolution less than the minimum supported by the LCD controller (64 in our example), this function will limit it to the minimum allowed by the hardware.

`fb_check_var()` also sets the appropriate RGB format. Our example uses 16 bits per pixel, and the controller maps each data word in the frame buffer into the commonly used RGB565 code: 5 bits for red, 6 bits for green, and 5 bits for blue. The offsets into the data word for each of the three colors are also set accordingly.

The `fb_set_par()` method configures the registers of the LCD controller depending on the values found in `fb_info.var`. This includes setting

- Horizontal sync duration, left margin, and right margin in `HSYNC_REG`
- Vertical sync duration, upper margin, and lower margin in `VSYNC_REG`
- The visible X and Y resolutions in `SIZE_REG`
- DMA parameters in `DMA_REG`

Assume that the GPS application attempts to alter the resolution of the QVGA display to 50x50. The following is the train of events:

1. The display is initially at QVGA resolution:

```
bash> fbset
mode "320x240-76"
    # D: 5.830 MHz, H: 18.219 kHz, V: 75.914 Hz
    geometry 320 240 320 240 16
    timings 171521 0 0 0 0 0 0
    rgba 5/11,6/5,5/0,0/0
```

```
endmode
```

2. The application does something like this:

```
struct fb_var_screeninfo vinfo;
fbfd = open("/dev/fb0", O_RDWR);
vinfo.xres = 50;
vinfo.yres = 50;
vinfo.bits_per_pixel = 8;

ioctl(fbfd, FBIOPUT_VSCREENINFO, &vinfo);
```

   Note that this is equivalent to the command `fbset -xres 50 -yres 50 -depth 8`.

3. The `FBIOPUT_VSCREENINFO` ioctl in the previous step triggers invocation of `myfb_check_var()`. This driver method expresses displeasure and rounds up the requested resolution to the minimum supported by the hardware, which is 64x64 in this case.

4. `myfb_set_par()` is invoked by the frame buffer core, which programs the new display parameters into LCD controller registers.

5. fbset now outputs new parameters:

```
bash> fbset
mode "64x64-1423"
    # D: 5.830 MHz, H: 91.097 kHz, V: 1423.386 Hz
    geometry 64 64 320 240 16
    timings 171521 0 0 0 0 0 0
    rgba 5/11,6/5,5/0,0/0
endmode
```

Color Modes

Common color modes supported by video hardware include *pseudo color* and *true color.* In the former, index numbers are mapped to RGB pixel encodings. By choosing a subset of available colors and by using the indices corresponding to the colors instead of the pixel values themselves, you can reduce demands on frame buffer memory. Your hardware needs to support this scheme of a modifiable color set (or *palette*), however.

In true color mode (which is what our example LCD controller supports), modifiable palettes are not relevant. However, you still have to satisfy the demands of the frame buffer console driver, which uses only 16 colors. For this, you have to create a pseudo palette by encoding the corresponding 16 raw RGB values into bits that can be directly fed to the hardware. This pseudo palette is stored in the `pseudo_palette` field of the `fb_info` structure. In Listing 12.2, `myfb_setcolreg()` populates it as follows:

```
((u32*)(info->pseudo_palette))[color_index] =
             (red << info->var.red.offset)     |
             (green << info->var.green.offset) |
             (blue << info->var.blue.offset)   |
             (transp << info->var.transp.offset);
```

Our LCD controller uses 16 bits per pixel and the RGB565 format, so as you saw earlier, the `fb_check_var()` method ensures that the red, green and blue values reside at bit offsets 11, 5, and 0, respectively. In addition

to the color index and the red, blue, and green values, `fb_setcolreg()` takes in an argument `transp`, to specify desired transparency effects. This mechanism, called *alpha blending*, combines the specified pixel value with the background color. The LCD controller in this example does not support alpha blending, so `myfb_check_var()` sets the `transp` offset and length to zero.

> The frame buffer abstraction is powerful enough to insulate applications from the characteristics of the display panel—whether it's RGB or BGR or something else. The red, blue, and green offsets set by `fb_check_var()` percolate to user space via the `fb_var_screeninfo` structure populated by the `FBIOGET_VSCREENINFO ioctl()`. Because applications such as X Windows are frame buffer-compliant, they paint pixels into the frame buffer according to the color offsets returned by this `ioctl()`.

Bit lengths used by the RGB encoding (5+6+5=16 in this case) is called the *color depth*, which is used by the frame buffer console driver to choose the logo file to display during boot (see the section "Boot Logo").

Screen Blanking

The `fb_blank()` method provides support for blanking and unblanking the display. This is mainly used for power management. To blank the navigation system's display after a 10-minute period of inactivity, do this:

```
bash> setterm -blank 10
```

This command percolates down the layers to the frame buffer layer and results in the invocation of `myfb_blank()`, which programs appropriate bits in `CTRL_REG`.

Accelerated Methods

If your user interface needs to perform heavy-duty video operations such as blending, stretching, moving bitmaps, or dynamic gradient generation, you likely require graphics acceleration to obtain acceptable performance. Let's briefly visit the `fb_ops` methods that you can leverage if your video hardware supports graphics acceleration.

The `fb_imageblit()` method draws an image to the display. This entry point provides an opportunity to your driver to leverage any special capabilities that your video controller might possess to hasten this operation. `cfb_imageblit()` is a generic library function provided by the frame buffer core to achieve this if you have nonaccelerated hardware. It's used, for instance, to output a logo to the screen during boot up. `fb_copyarea()` copies a rectangular area from one screen region to another. `cfb_copyarea()` provides an optimized way of doing this if your graphics controller does not possess any magic to accelerate this operation. The `fb_fillrect()` method speedily fills a rectangle with pixel lines. `cfb_fillrect()` offers a generic non-accelerated way to achieve this. The LCD controller in our navigation system does not provide for acceleration, so the example driver populates these methods using the generic software-optimized routines offered by the frame buffer core.

DirectFB

DirectFB (www.directfb.org) is a library built on top of the frame buffer interface that provides a simple window manager framework and hooks for hardware graphics acceleration and virtual interfaces that allow coexistence of multiple frame buffer applications. DirectFB, along with an accelerated frame buffer device driver downstream and a DirectFB-aware rendering engine such as Cairo (www.cairographics.org) upstream, is sometimes used on graphics-intensive embedded devices instead of more traditional solutions such as X Windows.

DMA from the Frame Buffer

The LCD controller in the navigation system contains a DMA engine that fetches picture frames from system memory. The controller dispatches the obtained graphics data to the display panel. The rate of DMA sustains the refresh rate of the display. A non-cacheable frame buffer suitable for coherent access is allocated using `dma_alloc_coherent()` from `myfb_probe()`. (We discussed coherent DMA mapping in Chapter 10, "Peripheral Component Interconnect.") `myfb_set_par()` writes this allocated DMA address to the `DMA_REG` register in the LCD controller.

When the driver enables DMA by calling `myfb_enable_controller()`, the controller starts ferrying pixel data from the frame buffer to the display using synchronous DMA. So, when the GPS application maps the frame buffer (using `mmap()`) and writes location information to it, the pixels gets painted onto the LCD.

Contrast and Backlight

The LCD controller in the navigation system supports contrast control using the `CONTRAST_REG` register. The driver exports this to user space via `myfb_ioctl()`. The GPS application controls contrast as follows:

```
unsigned int my_fd, desired_contrast_level = 100;
/* Open the frame buffer */
my_fd = open("/dev/fb0", O_RDWR);
ioctl(my_fd, MYFB_SET_BRIGHTNESS, &desired_contrast_level);
```

The LCD panel on the navigation system is illuminated using a backlight. The processor controls the backlight inverter through GPIO lines, so you can turn the light on or off by wiggling the corresponding pins. The kernel abstracts a generic backlight interface via sysfs nodes. To tie with this interface, your driver has to populate a `backlight_ops` structure with methods for obtaining and updating backlight brightness, and register it with the kernel using `backlight_device_register()`. Look inside *drivers/video/backlight/* for the backlight interface sources and recursively grep the *drivers/* tree for `backlight_device_register()` to locate video drivers that use this interface. Listing 12.2 does not implement backlight manipulation operations.

Listing 12.2. Frame Buffer Driver for the Navigation System

Code View:

```
#include <linux/fb.h>
#include <linux/dma-mapping.h>
#include <linux/platform_device.h>

/* Address map of LCD controller registers */
#define LCD_CONTROLLER_BASE   0x01000D00
#define SIZE_REG     (*(volatile u32 *)(LCD_CONTROLLER_BASE))
#define HSYNC_REG    (*(volatile u32 *)(LCD_CONTROLLER_BASE + 4))
#define VSYNC_REG    (*(volatile u32 *)(LCD_CONTROLLER_BASE + 8))
#define CONF_REG     (*(volatile u32 *)(LCD_CONTROLLER_BASE + 12))
```

```
#define CTRL_REG     (*(volatile u32 *)(LCD_CONTROLLER_BASE + 16))
#define DMA_REG      (*(volatile u32 *)(LCD_CONTROLLER_BASE + 20))
#define STATUS_REG   (*(volatile u32 *)(LCD_CONTROLLER_BASE + 24))
#define CONTRAST_REG (*(volatile u32 *)(LCD_CONTROLLER_BASE + 28))
#define LCD_CONTROLLER_SIZE   32

/* Resources for the LCD controller platform device */
static struct resource myfb_resources[] = {
  [0] = {
    .start      = LCD_CONTROLLER_BASE,
    .end        = LCD_CONTROLLER_SIZE,
    .flags      = IORESOURCE_MEM,
  },
};

/* Platform device definition */
static struct platform_device myfb_device = {
  .name      = "myfb",
  .id        = 0,
  .dev       = {
    .coherent_dma_mask = 0xffffffff,
  },
  .num_resources = ARRAY_SIZE(myfb_resources),
  .resource      = myfb_resources,
};

/* Set LCD controller parameters */
static int
myfb_set_par(struct fb_info *info)
{
  unsigned long adjusted_fb_start;
  struct fb_var_screeninfo *var = &info->var;
  struct fb_fix_screeninfo *fix = &info->fix;

  /* Top 16 bits of HSYNC_REG hold HSYNC duration, next 8 contain
     the left margin, while the bottom 8 house the right margin */
  HSYNC_REG = (var->hsync_len << 16) |
              (var->left_margin << 8)|
              (var->right_margin);
  /* Top 16 bits of VSYNC_REG hold VSYNC duration, next 8 contain
     the upper margin, while the bottom 8 house the lower margin */
  VSYNC_REG = (var->vsync_len << 16)  |
              (var->upper_margin << 8)|
              (var->lower_margin);

  /* Top 16 bits of SIZE_REG hold xres, bottom 16 hold yres */
  SIZE_REG  = (var->xres << 16) | (var->yres);

  /* Set bits per pixel, pixel polarity, clock dividers for
     the pixclock, and color/monochrome mode in CONF_REG */
  /* ... */

  /* Fill DMA_REG with the start address of the frame buffer
     coherently allocated from myfb_probe(). Adjust this address
     to account for any offset to the start of screen area */
  adjusted_fb_start = fix->smem_start +
          (var->yoffset * var->xres_virtual + var->xoffset) *
          (var->bits_per_pixel) / 8;
  __raw_writel(adjusted_fb_start, (unsigned long *)DMA_REG);
```

```
  /*  Set the DMA burst length and watermark sizes in DMA_REG */
  /* ... */

  /* Set fixed information */
  fix->accel  = FB_ACCEL_NONE;       /* No hardware acceleration */
  fix->visual = FB_VISUAL_TRUECOLOR; /* True color mode */
  fix->line_length = var->xres_virtual * var->bits_per_pixel/8;

  return 0;
}

/* Enable LCD controller */
static void
myfb_enable_controller(struct fb_info *info)
{
  /* Enable LCD controller, start DMA, enable clocks and power
     by writing to CTRL_REG */
  /* ... */
}
/* Disable LCD controller */
static void
myfb_disable_controller(struct fb_info *info)
{
  /* Disable LCD controller, stop DMA, disable clocks and power
     by writing to CTRL_REG */
  /* ... */
}

/* Sanity check and adjustment of variables */
static int
myfb_check_var(struct fb_var_screeninfo *var, struct fb_info *info)
{
  /* Round up to the minimum resolution supported by
     the LCD controller */
  if (var->xres < 64) var->xres = 64;
  if (var->yres < 64) var->yres = 64;

  /* ... */
  /* This hardware supports the RGB565 color format.
     See the section "Color Modes" for more details */
  if (var->bits_per_pixel == 16) {
    /* Encoding Red */
    var->red.length = 5;
    var->red.offset = 11;
    /* Encoding Green */
    var->green.length = 6;
    var->green.offset = 5;
    /* Encoding Blue */
    var->blue.length = 5;
    var->blue.offset = 0;
    /* No hardware support for alpha blending */
    var->transp.length = 0;
    var->transp.offset = 0;
  }
  return 0;
}

/* Blank/unblank screen */
static int
```

```
myfb_blank(int blank_mode, struct fb_info *info)
{
  switch (blank_mode) {
  case FB_BLANK_POWERDOWN:
  case FB_BLANK_VSYNC_SUSPEND:
  case FB_BLANK_HSYNC_SUSPEND:
  case FB_BLANK_NORMAL:
    myfb_disable_controller(info);
    break;
  case FB_BLANK_UNBLANK:
    myfb_enable_controller(info);
    break;
  }
  return 0;
}

/* Configure pseudo color palette map */
static int
myfb_setcolreg(u_int color_index, u_int red, u_int green,
               u_int blue, u_int transp, struct fb_info *info)
{
  if (info->fix.visual == FB_VISUAL_TRUECOLOR) {
    /* Do any required translations to convert red, blue, green and
       transp, to values that can be directly fed to the hardware */
    /* ... */

    ((u32 *)(info->pseudo_palette))[color_index] =
           (red << info->var.red.offset)     |
           (green << info->var.green.offset) |
           (blue << info->var.blue.offset)   |
           (transp << info->var.transp.offset);
  }
  return 0;
}

/* Device-specific ioctl definition */
#define MYFB_SET_BRIGHTNESS _IOW('M', 3, int8_t)

/* Device-specific ioctl */
static int
myfb_ioctl(struct fb_info *info, unsigned int cmd,
           unsigned long arg)
{
  u32 blevel ;
  switch (cmd) {
    case MYFB_SET_BRIGHTNESS :
      copy_from_user((void *)&blevel, (void *)arg,
                     sizeof(blevel)) ;
      /* Write blevel to CONTRAST_REG */
      /* ... */
      break;
    default:
      return –EINVAL;
  }
  return 0;
}

/* The fb_ops structure */
static struct fb_ops myfb_ops = {
```

```
  .owner        = THIS_MODULE,
  .fb_check_var = myfb_check_var,/* Sanity check */
  .fb_set_par   = myfb_set_par,  /* Program controller registers */
  .fb_setcolreg = myfb_setcolreg,/* Set color map */
  .fb_blank     = myfb_blank,    /* Blank/unblank display */
  .fb_fillrect  = cfb_fillrect,  /* Generic function to fill
                                    rectangle */
  .fb_copyarea  = cfb_copyarea,  /* Generic function to copy area */
  .fb_imageblit = cfb_imageblit, /* Generic function to draw */
  .fb_ioctl     = myfb_ioctl,    /* Device-specific ioctl */
};

/* Platform driver's probe() routine */
static int __init
myfb_probe(struct platform_device *pdev)
{
  struct fb_info *info;
  struct resource *res;

  info = framebuffer_alloc(0, &pdev->dev);
  /* ... */
  /* Obtain the associated resource defined while registering the
     corresponding platform_device */
  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
  /* Get the kernel's sanction for using the I/O memory chunk
     starting from LCD_CONTROLLER_BASE and having a size of
     LCD_CONTROLLER_SIZE bytes */
  res = request_mem_region(res->start, res->end - res->start + 1,
                           pdev->name);

  /* Fill the fb_info structure with fixed (info->fix) and variable
     (info->var) values such as frame buffer length, xres, yres,
     bits_per_pixel, fbops, cmap, etc */
  initialize_fb_info(info, pdev);  /* Not expanded */
  info->fbops = &myfb_ops;
  fb_alloc_cmap(&info->cmap, 16, 0);

  /* DMA-map the frame buffer memory coherently. info->screen_base
     holds the CPU address of the mapped buffer,
     info->fix.smem_start carries the associated hardware address */
  info->screen_base = dma_alloc_coherent(0, info->fix.smem_len,
                                  (dma_addr_t *)&info->fix.smem_start,
                                   GFP_DMA | GFP_KERNEL);
  /* Set the information in info->var to the appropriate
     LCD controller registers */
  myfb_set_par(info);

  /* Register with the frame buffer core */
  register_framebuffer(info);
  return 0;
}

/* Platform driver's remove() routine */
static int
myfb_remove(struct platform_device *pdev)
{
  struct fb_info *info = platform_get_drvdata(pdev);
  struct resource *res;
```

```
  /* Disable screen refresh, turn off DMA,.. */
  myfb_disable_controller(info);

  /* Unregister frame buffer driver */
  unregister_framebuffer(info);
  /* Deallocate color map */
  fb_dealloc_cmap(&info->cmap);
  kfree(info->pseudo_palette);

  /* Reverse of framebuffer_alloc() */
  framebuffer_release(info);
  /* Release memory region */
  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
  release_mem_region(res->start, res->end - res->start + 1);
  platform_set_drvdata(pdev, NULL);

  return 0;
}

/* The platform driver structure */
static struct platform_driver myfb_driver = {
  .probe     = myfb_probe,
  .remove    = myfb_remove,
  .driver    = {
    .name    = "myfb",
  },
};

/* Module Initialization */
int __init
myfb_init(void)
{
  platform_device_add(&myfb_device);
  return platform_driver_register(&myfb_driver);
}

/* Module Exit */
void __exit
myfb_exit(void)
{
  platform_driver_unregister(&myfb_driver);
  platform_device_unregister(&myfb_device);
}

module_init(myfb_init);
module_exit(myfb_exit);
```

Console Drivers

A *console* is a device that displays `printk()` messages generated by the kernel. If you look at Figure 12.5, you can see that console drivers lie in two tiers: a top level constituting drivers such as the virtual terminal driver, the printer console driver, and the example `USB_UART` console driver (discussed soon), and bottom-level drivers that are responsible for advanced operations. Consequently, there are two main interface definition structures used by console drivers. Top-level console drivers revolve around `struct console`, which defines basic operations such as `setup()` and `write()`. Bottom-level drivers center on `struct consw` that specifies advanced operations such as setting cursor properties, console switching, blanking, resizing, and setting palette information. These structures are defined in *include/linux/console.h* as follows:

1.
```
struct console {
  char  name[8];
  void  (*write)(struct console *, const char *, unsigned);
  int   (*read)(struct console *, char *, unsigned);
  /* ... */
  void  (*unblank)(void);
  int   (*setup)(struct console *, char *);
  /* ... */
};
```

2.
```
struct consw {
  struct module *owner;
  const char *(*con_startup)(void);
  void        (*con_init)(struct vc_data *, int);
  void        (*con_deinit)(struct vc_data *);
  void        (*con_clear)(struct vc_data *, int, int, int, int);
  void        (*con_putc)(struct vc_data *, int, int, int);
  void        (*con_putcs)(struct vc_data *,
                           const unsigned short *, int, int, int);
  void        (*con_cursor)(struct vc_data *, int);
  int         (*con_scroll)(struct vc_data *, int, int, int, int);
  /* ... */
};
```

As you might have guessed by looking at Figure 12.5, most console devices need both levels of drivers working in tandem. The *vt* driver is the top-level console driver in many situations. On PC-compatible systems, the VGA console driver (*vgacon*) is usually the bottom-level console driver; whereas on embedded devices, the frame buffer console driver (*fbcon*) is often the bottom-level driver. Because of the indirection offered by the frame buffer abstraction, fbcon, unlike other bottom-level console drivers, is hardware-independent.

Let's briefly look at the architecture of both levels of console drivers:

- The top-level driver populates a `struct console` with prescribed entry points and registers it with the kernel using `register_console()`. Unregistering is accomplished using `unregister_console()`. This is the driver that interacts with `printk()`. The entry points belonging to this driver call on the services of the associated bottom-level console driver.

- The bottom-level console driver populates a `struct consw` with specified entry points and registers it with the kernel using `register_con_driver()`. Unregistering is done using `unregister_con_driver()`. When the system supports multiple console drivers, the driver might instead invoke `take_over_console()` to register itself and take over the existing console. `give_up_console()` accomplishes the reverse. For conventional displays, bottom-level drivers interact with the top-level vt console driver and the *vc_screen* character driver that allows access to virtual console memory.

Some simple consoles, such as line printers and the `USB_UART` discussed next, need only a top-level console driver.

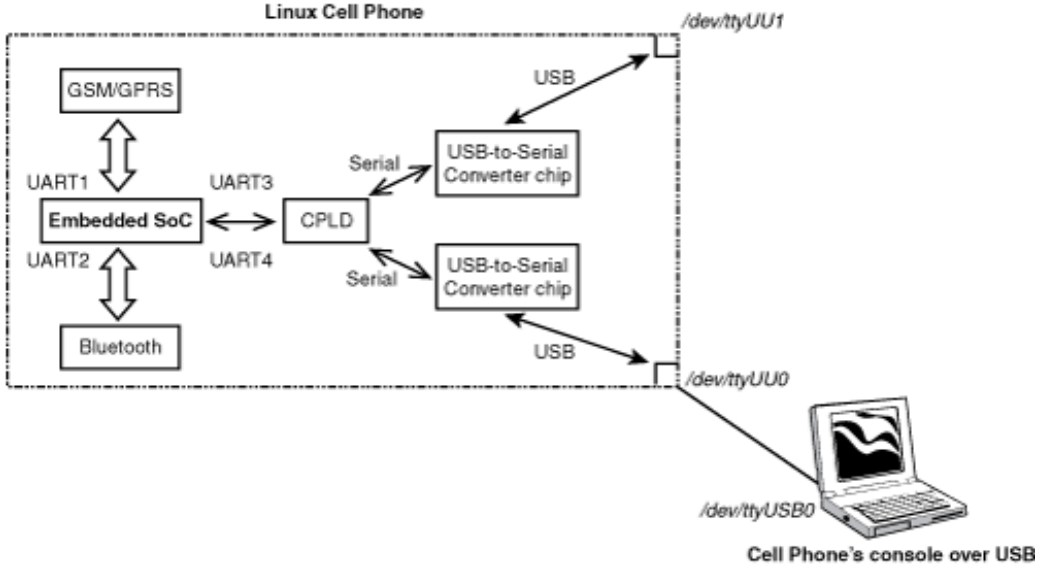
The fbcon driver in the 2.6 kernel also supports console rotation. Display panels on PDAs and cell phones are usually mounted in portrait orientation, whereas automotive dashboards and IP phones are examples of systems where the display panel is likely to be in landscape mode. Sometimes, due to economics or other factors, an embedded device may require a landscape LCD to be mounted in portrait mode or vice versa. Console rotation support comes handy in such situations. Because fbcon is hardware-independent, the console rotation implementation is also generic. To enable console rotation, enable `CONFIG_FRAMEBUFFER_CONSOLE_ROTATION` during kernel configuration and add `fbcon=rotate:X` to the kernel command line, where `X` is 0 for normal orientation, 1 for 90-degree rotation, 2 for 180-degree rotation, and 3 for 270-degree rotation.

Device Example: Cell Phone Revisited

To learn how to write console drivers, let's revisit the Linux cell phone that we used in Chapter 6. Our task in this section is to develop a console driver that operates over the `USB_UART`s in the cell phone. For convenience, Figure 12.7 reproduces the cell phone from Figure 6.5 in Chapter 6. Let's write a console driver that gets `printk()` messages out of the door via a `USB_UART`. The messages are picked up by a PC host and displayed to the user via a terminal emulator session.

Figure 12.7. Console over **USB_UART**.

[View full size image]



Listing 12.3 develops the console driver that works over the `USB_UART`s. The `usb_uart_port[]` structure and a

few definitions used by the USB_UART driver in Chapter 6 are included in this listing, too, to create a complete driver. Comments associated with the listing explain the driver's operation.

Figure 12.7 shows the position of our example USB_UART console driver within the Linux-Video subsystem. As you can see, the USB_UART is a simple device that needs only a top-level console driver.

Listing 12.3. Console over **USB_UART**

Code View:

```
#include <linux/console.h>
#include <linux/serial_core.h>
#include <asm/io.h>

#define USB_UART_PORTS          2          /* The cell phone has 2
                                              USB_UART ports */
/* Each USB_UART has a 3-byte register set consisting of
   UU_STATUS_REGISTER at offset 0, UU_READ_DATA_REGISTER at
   offset 1, and UU_WRITE_DATA_REGISTER at offset 2, as shown
   in Table One of Chapter 6, "Serial Drivers" */
#define USB_UART1_BASE           0xe8000000 /* Memory base for USB_UART1 */
#define USB_UART2_BASE           0xe9000000 /* Memory base for USB_UART1 */
#define USB_UART_REGISTER_SPACE  0x3
/* Semantics of bits in the status register */
#define USB_UART_TX_FULL         0x20
#define USB_UART_RX_EMPTY        0x10
#define USB_UART_STATUS          0x0F

#define USB_UART1_IRQ            3
#define USB_UART2_IRQ            4
#define USB_UART_CLK_FREQ        16000000
#define USB_UART_FIFO_SIZE       32

/* Parameters of each supported USB_UART port */
static struct uart_port usb_uart_port[] = {
  {
    .mapbase  = (unsigned int)USB_UART1_BASE,
    .iotype   = UPIO_MEM,             /* Memory mapped */
    .irq      = USB_UART1_IRQ,        /* IRQ */
    .uartclk  = USB_UART_CLK_FREQ,    /* Clock HZ */
    .fifosize = USB_UART_FIFO_SIZE,   /* Size of the FIFO */
    .flags    = UPF_BOOT_AUTOCONF,    /* UART port flag */
    .line     = 0,                    /* UART Line number */
  },
  {
    .mapbase   = (unsigned int)USB_UART2_BASE,
    .iotype    = UPIO_MEM,            /* Memory mapped */
    .irq       = USB_UART2_IRQ,       /* IRQ */
    .uartclk   = USB_UART_CLK_FREQ,   /* CLock HZ */
    .fifosize  = USB_UART_FIFO_SIZE,  /* Size of the FIFO */
    .flags     = UPF_BOOT_AUTOCONF,   /* UART port flag */
    .line      = 1,                   /* UART Line number */
  }
};

/* Write a character to the USB_UART port */
static void
usb_uart_putc(struct uart_port *port, unsigned char c)
{
```

```
  /* Wait until there is space in the TX FIFO of the USB_UART.
     Sense this by looking at the USB_UART_TX_FULL
     bit in the status register */
  while (__raw_readb(port->membase) & USB_UART_TX_FULL);

  /* Write the character to the data port*/
  __raw_writeb(c, (port->membase+1));
}

/* Console write */
static void
usb_uart_console_write(struct console *co, const char *s,
                           u_int count)
{
  int i;

  /* Write each character */
  for (i = 0; i < count; i++, s++) {
    usb_uart_putc(&usb_uart_port[co->index], *s);
  }
}

/* Get communication parameters */
static void __init
usb_uart_console_get_options(struct uart_port *port,
                             int *baud, int *parity, int *bits)
{
  /* Read the current settings (possibly set by a bootloader)
     or return default values for parity, number of data bits,
     and baud rate */
  *parity = 'n';
  *bits = 8;
  *baud = 115200;
}

/* Setup console communication parameters */
static int __init
usb_uart_console_setup(struct console *co, char *options)
{
  struct uart_port *port;
  int baud, bits, parity, flow;

  /* Validate port number and get a handle to the
     appropriate structure */
  if (co->index == -1 || co->index >= USB_UART_PORTS) {
    co->index = 0;
  }
  port = &usb_uart_port[co->index];

  /* Use functions offered by the serial layer to parse options */
  if (options) {
    uart_parse_options(options, &baud, &parity, &bits, &flow);
  } else {
    usb_uart_console_get_options(port, &baud, &parity, &bits);
  }
  return uart_set_options(port, co, baud, parity, bits, flow);
}

/* Populate the console structure */
```

```
static struct console usb_uart_console = {
  .name    = "ttyUU",                /* Console name */
  .write   = usb_uart_console_write,  /* How to printk to the
                                         console */
  .device  = uart_console_device,     /* Provided by the serial core */
  .setup   = usb_uart_console_setup,  /* How to setup the console */
  .flags   = CON_PRINTBUFFER,         /* Default flag */
  .index   = -1,                      /* Init to invalid value */
};


/* Console Initialization */
static int __init
usb_uart_console_init(void)
{
  /* ... */

  /* Register this console */
  register_console(&usb_uart_console);

  return 0;
}

console_initcall(usb_uart_console_init); /* Mark console init */
```

After this driver has been built as part of the kernel, you can activate it by appending `console=ttyUUX` (where `X` is 0 or 1) to the kernel command line.

Boot Logo

A popular feature offered by the frame buffer subsystem is the boot logo. To display a logo, enable `CONFIG_LOGO` during kernel configuration and select an available logo. You may also add a custom logo image in the *drivers/video/logo/* directory.

CLUT224 is a commonly used boot logo image format that supports 224 colors. The working of this format is similar to pseudo palettes described in the section "Color Modes." A CLUT224 image is a C file containing two structures:

- A CLUT (*Color Look Up Table*), which is a character array of 224 RGB tuples (thus having a size of 224*3 bytes). Each 3-byte CLUT element is a combination of red, green, and blue colors.

- A data array whose each byte is an index into the CLUT. The indices start at 32 and extend until 255 (thus supporting 224 colors). Index 32 refers to the first element in the CLUT. The logo manipulation code (in *drivers/video/fbmem.c*) creates frame buffer pixel data from the CLUT tuple corresponding to each index in the data array. Image display is accomplished using the low-level frame buffer driver's `fb_imageblit()` method, as indicated in the section "Accelerated Methods."

Other supported logo formats are the 16-color *vga16* and the black-and-white *mono*. Scripts are available in the top-level *scripts/* directory to convert standard *Portable Pixel Map* (PPM) files to the supported logo formats.

If the frame buffer device is also the console, boot messages scroll under the logo. You may prefer to disable console messages on production-level systems (by adding `console=/dev/null` to the kernel command line) and display a customer-supplied CLUT224 "splash screen" image as the boot logo.

Debugging

The virtual frame buffer driver, enabled by setting `CONFIG_FB_VIRTUAL` in the configuration menu, operates over a pseudo graphics adapter. You can use this driver's assistance to debug the frame buffer subsystem.

Some frame buffer drivers, such as *intelfb*, offer an additional configuration option that you may enable to generate driver-specific debug information.

To discuss issues related to frame buffer drivers, subscribe to the linux-fbdev-devel mailing list, https://lists.sourceforge.net/lists/listinfo/linux-fbdev-devel/.

Debugging console drivers is not an easy job because you can't call `printk()` from inside the driver. If you have a spare console device such as a serial port, you can implement a UART/tty form factor of your console driver first (as we did in Chapter 6 for the USB_UART device used in this chapter) and debug that driver by operating on */dev/tty* and printing messages to the spare console. You can then repackage the debugged code regions in the form of a console driver.

Looking at the Sources

The frame buffer core layer and low-level frame buffer drivers reside in the *drivers/video/* directory. Generic frame buffer structures are defined in *include/linux/fb.h*, whereas chipset-specific headers stay inside *include/video/.* The fbmem driver, *drivers/video/fbmem.c*, creates the */dev/fbX* character devices and is the front end for handling frame buffer ioctl commands issued by user applications.

The intelfb driver, *drivers/video/intelfb/**, is the low-level frame buffer driver for several Intel graphics controllers such as the one integrated with the 855 GME North Bridge. The radeonfb driver, *drivers/video/aty/**, is the frame buffer driver for Radeon Mobility AGP graphics hardware from ATI technologies. The source files, *drivers/video/*fb.c*, are all frame buffer drivers for graphics controllers, including those integrated into several SoCs. You can use *drivers/video/skeletonfb.c* as the starting point if you are writing a custom low-level frame buffer driver. Look at *Documentation/fb/** for more documentation on the frame buffer layer.

The home page of the Linux frame buffer project is www.linux-fbdev.org. This website contains HOWTOs, links to frame buffer drivers and utilities, and pointers to related web pages.

Console drivers, both frame buffer-based and otherwise, live inside *drivers/video/console/.* To find out how `printk()` logs kernel messages to an internal buffer and calls console drivers, look at *kernel/printk.c*.

Table 12.2 contains the main data structures used in this chapter and their location in the source tree. Table 12.3 lists the main kernel programming interfaces that you used in this chapter with the location of their definitions.

Table 12.2. Summary of Data Structures

Data Structure	Location	Description
`fb_info`	*include/linux/fb.h*	Central data structure used by low-level frame buffer drivers
`fb_ops`	*include/linux/fb.h*	Contains addresses of all entry points provided by low-level frame buffer drivers
`fb_var_screeninfo`	*include/linux/fb.h*	Contains variable information pertaining to video hardware such as the X-resolution, Y-resolution, and `HYSNC/VSYNC` durations
`fb_fix_screeninfo`	*include/linux/fb.h*	Fixed information about video hardware such as the start address of the frame buffer
`fb_cmap`	*include/linux/fb.h*	The RGB color map for a frame buffer device
`console`	*include/linux/console.h*	Representation of a top-level console driver
`consw`	*include/linux/console.h*	Representation of a bottom-level console driver

Table 12.3. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`register_framebuffer()`	*drivers/video/fbmem.c*	Registers a low-level frame buffer device.
`unregister_framebuffer()`	*drivers/video/fbmem.c*	Releases a low-level frame buffer device.
`framebuffer_alloc()`	*drivers/video/fbsysfs.c*	Allocates memory for an `fb_info` structure.
`framebuffer_release()`	*drivers/video/fbsysfs.c*	Reverse of `framebuffer_alloc()`.
`fb_alloc_cmap()`	*drivers/video/fbcmap.c*	Allocates color map.
`fb_dealloc_cmap()`	*drivers/video/fbcmap.c*	Frees color map.
`dma_alloc_coherent()`	*include/asm-generic/dma-mapping.h*	Allocates and maps a coherent DMA buffer. See `pci_alloc_consistent()` in Chapter 10.
`dma_free_coherent()`	*include/asm-generic/dma-mapping.h*	Frees a coherent DMA buffer. See `pci_free_consistent()` in Chapter 10.
`register_console()`	*kernel/printk.c*	Registers a top-level console driver.
`unregister_console()`	*kernel/printk.c*	Unregisters a top-level console driver.
`register_con_driver()` `take_over_console()`	*drivers/char/vt.c*	Registers/binds a bottom-level console driver.
`unregister_con_driver()` `give_up_console()`	*drivers/char/vt.c*	Unregisters/unbinds a bottom-level console driver.

Chapter 13. Audio Drivers

In This Chapter

- Audio Architecture 392
- Linux-Sound Subsystem 394
- Device Example: MP3 Player 396
- Debugging 412
- Looking at the Sources 412

Audio hardware provides computer systems the capability to generate and capture sound. Audio is an integral component in both the PC and the embedded space, for chatting on a laptop, making a call from a cell phone, listening to an MP3 player, streaming multimedia from a set-top box, or announcing instructions on a medical-grade system. If you run Linux on any of these devices, you need the services offered by the Linux-Sound subsystem.

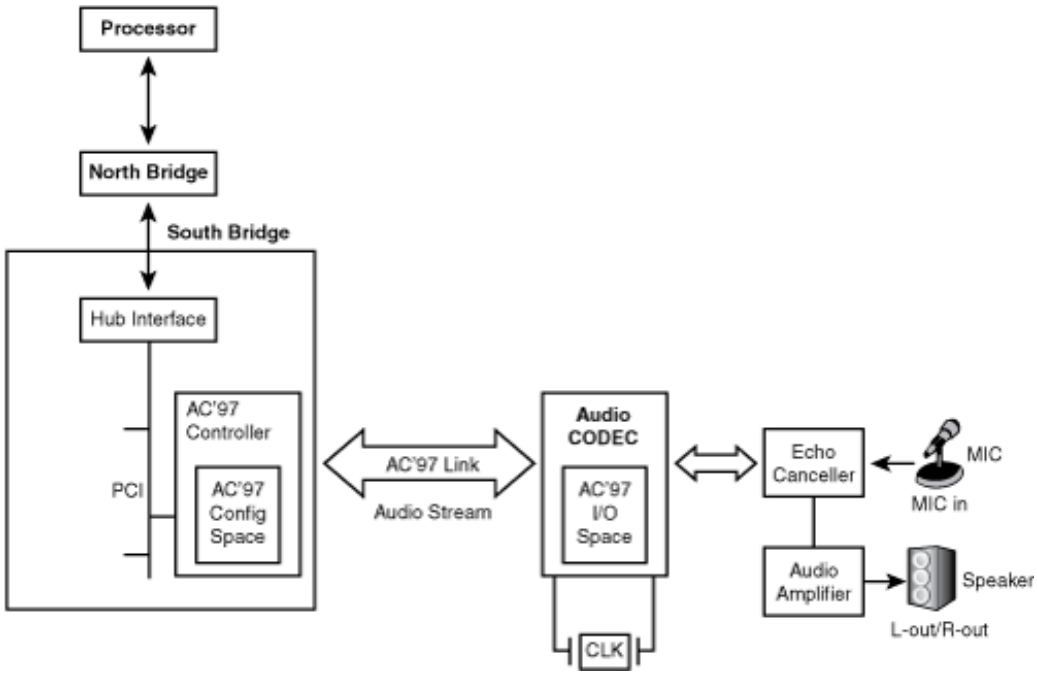
In this chapter, let's find out how the kernel supports audio controllers and codecs. Let's learn the architecture of the Linux-Sound subsystem and the programming model that it exports.

Audio Architecture

Figure 13.1 shows audio connection on a PC-compatible system. The audio controller on the South Bridge, together with an external codec, interfaces with analog audio circuitry.

Figure 13.1. Audio in the PC environment.

[View full size image]



An *audio codec* converts digital audio data to analog sound signals for playing through speakers and performs the reverse operation for recording through a microphone. Other common audio inputs and outputs that interface with a codec include headsets, earphones, handsets, hands-free, line-in, and line-out. A codec also offers *mixer* functionality that connects it to a combination of these audio inputs and outputs, and controls the volume gain of associated audio signals.[1]

[1] This definition of a mixer is from a software perspective. *Sound mixing* or *data mixing* refers to the capability of some codecs to mix multiple sound streams and generate a single stream. This is needed, for example, if you want to superimpose an announcement while a voice communication is in progress on an IP phone. The alsa-lib library, discussed in the latter part of this chapter, supports a plug-in feature called *dmix* that performs data mixing in software if your codec does not have the capability to perform this operation in hardware.

Digital audio data is obtained by sampling analog audio signals at specific bit rates using a technique called *pulse code modulation* (PCM). CD quality is, for example, sound sampled at 44.1KHz, using 16 bits to hold each sample. A codec is responsible for recording audio by sampling at supported PCM bit rates and for playing audio originally sampled at different PCM bit rates.

A sound card may support one or more codecs. Each codec may, in turn, support one or more audio substreams in mono or stereo.

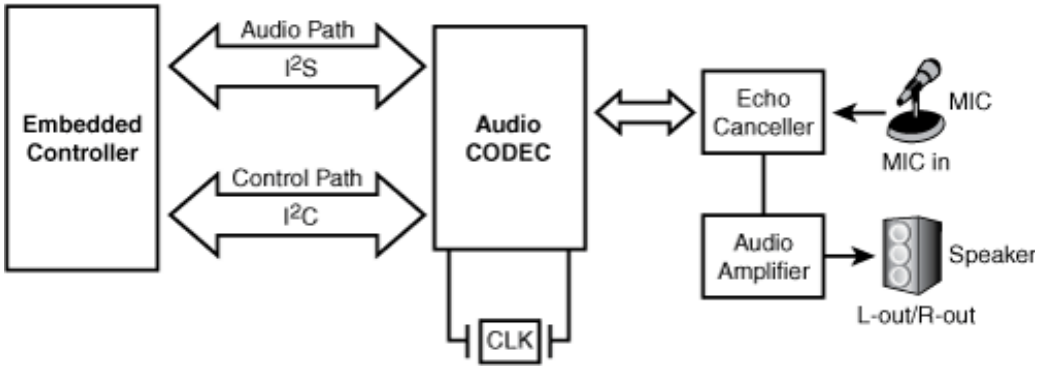
The Audio Codec'97 (AC'97) and the Inter-IC Sound (I^2S) bus are examples of industry standard interfaces that connect audio controllers to codecs:

- The Intel AC'97 specification, downloadable from http://download.intel.com/, specifies the semantics and locations of audio registers. Configuration registers are part of the audio controller, while the I/O register space is situated inside the codec. Requests to operate on I/O registers are forwarded by the audio controller to the codec over the AC'97 link. The register that controls line-in volume, for example, lives at offset `0x10` within the AC'97 I/O space. The PC system in Figure 13.1 uses AC'97 to communicate with an external codec.

- The I^2S specification, downloadable from www.nxp.com/acrobat_download/various/I2SBUS.pdf, is a codec

interface standard developed by Philips. The embedded device shown in Figure 13.2 uses I^2S to send audio data to the codec. Programming the codec's I/O registers is done via the I^2C bus.

Figure 13.2. Audio connection on an embedded system.



AC'97 has limitations pertaining to the number of supported channels and bit rates. Recent South Bridge chipsets from Intel feature a new technology called High Definition (HD) Audio that offers higher-quality, surround sound, and multistreaming capabilities.

Chapter 13. Audio Drivers

In This Chapter

- Audio Architecture 392
- Linux-Sound Subsystem 394
- Device Example: MP3 Player 396
- Debugging 412
- Looking at the Sources 412

Audio hardware provides computer systems the capability to generate and capture sound. Audio is an integral component in both the PC and the embedded space, for chatting on a laptop, making a call from a cell phone, listening to an MP3 player, streaming multimedia from a set-top box, or announcing instructions on a medical-grade system. If you run Linux on any of these devices, you need the services offered by the Linux-Sound subsystem.

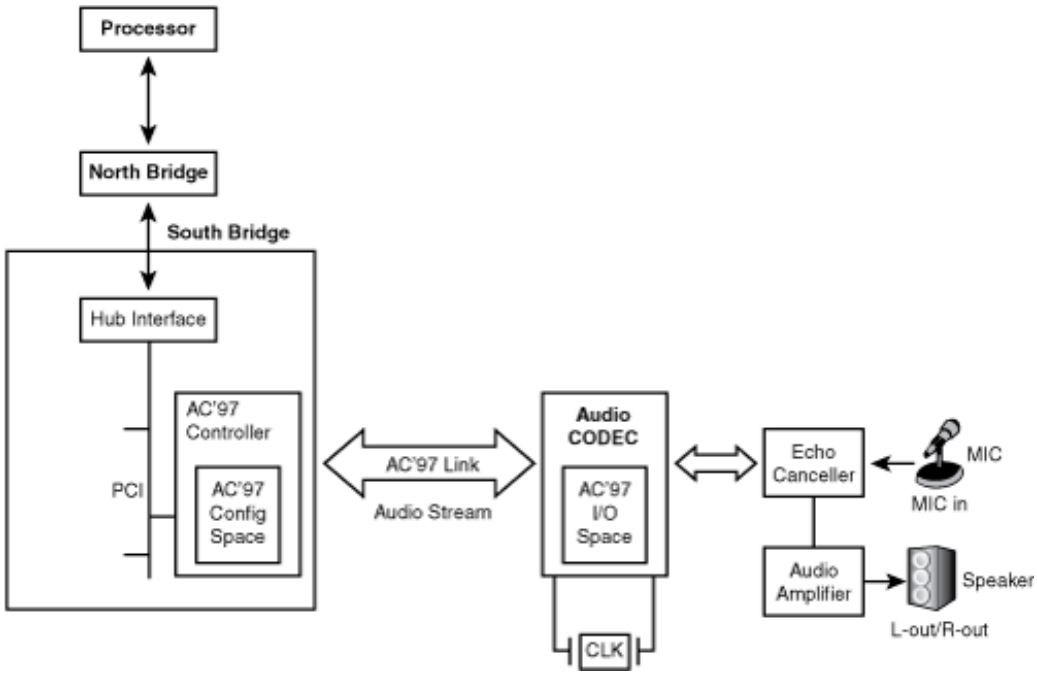
In this chapter, let's find out how the kernel supports audio controllers and codecs. Let's learn the architecture of the Linux-Sound subsystem and the programming model that it exports.

Audio Architecture

Figure 13.1 shows audio connection on a PC-compatible system. The audio controller on the South Bridge, together with an external codec, interfaces with analog audio circuitry.

Figure 13.1. Audio in the PC environment.

[View full size image]



An *audio codec* converts digital audio data to analog sound signals for playing through speakers and performs the reverse operation for recording through a microphone. Other common audio inputs and outputs that interface with a codec include headsets, earphones, handsets, hands-free, line-in, and line-out. A codec also offers *mixer* functionality that connects it to a combination of these audio inputs and outputs, and controls the volume gain of associated audio signals.[1]

[1] This definition of a mixer is from a software perspective. *Sound mixing* or *data mixing* refers to the capability of some codecs to mix multiple sound streams and generate a single stream. This is needed, for example, if you want to superimpose an announcement while a voice communication is in progress on an IP phone. The alsa-lib library, discussed in the latter part of this chapter, supports a plug-in feature called *dmix* that performs data mixing in software if your codec does not have the capability to perform this operation in hardware.

Digital audio data is obtained by sampling analog audio signals at specific bit rates using a technique called *pulse code modulation* (PCM). CD quality is, for example, sound sampled at 44.1KHz, using 16 bits to hold each sample. A codec is responsible for recording audio by sampling at supported PCM bit rates and for playing audio originally sampled at different PCM bit rates.

A sound card may support one or more codecs. Each codec may, in turn, support one or more audio substreams in mono or stereo.

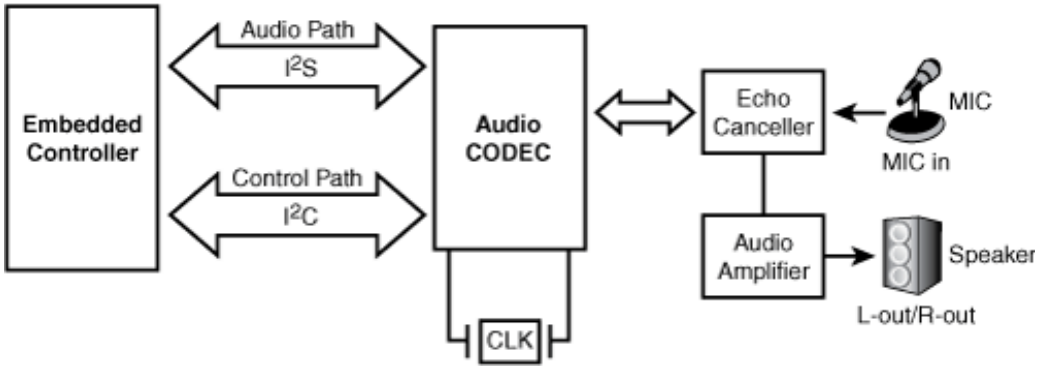
The Audio Codec'97 (AC'97) and the Inter-IC Sound (I^2S) bus are examples of industry standard interfaces that connect audio controllers to codecs:

- The Intel AC'97 specification, downloadable from http://download.intel.com/, specifies the semantics and locations of audio registers. Configuration registers are part of the audio controller, while the I/O register space is situated inside the codec. Requests to operate on I/O registers are forwarded by the audio controller to the codec over the AC'97 link. The register that controls line-in volume, for example, lives at offset `0x10` within the AC'97 I/O space. The PC system in Figure 13.1 uses AC'97 to communicate with an external codec.

- The I^2S specification, downloadable from www.nxp.com/acrobat_download/various/I2SBUS.pdf, is a codec

interface standard developed by Philips. The embedded device shown in Figure 13.2 uses I^2S to send audio data to the codec. Programming the codec's I/O registers is done via the I^2C bus.

Figure 13.2. Audio connection on an embedded system.



AC'97 has limitations pertaining to the number of supported channels and bit rates. Recent South Bridge chipsets from Intel feature a new technology called High Definition (HD) Audio that offers higher-quality, surround sound, and multistreaming capabilities.

Linux-Sound Subsystem

Advanced Linux Sound Architecture (ALSA) is the sound subsystem of choice in the 2.6 kernel. *Open Sound System* (OSS), the sound layer in the 2.4 kernel, is now obsolete and depreciated. To help the transition from OSS to ALSA, the latter provides OSS emulation that allows applications conforming to the OSS API to run unchanged over ALSA. Linux-Sound frameworks such as ALSA and OSS render audio applications independent of the underlying hardware, just as codec standards such as AC'97 and I^2S do away with the need of writing separate audio drivers for each sound card.

Take a look at Figure 13.3 to understand the architecture of the Linux-Sound subsystem. The constituent pieces of the subsystem are as follows:

- The sound core, which is a code base consisting of routines and structures available to other components of the Linux-Sound layer. Like the core layers belonging to other driver subsystems, the sound core provides a level of indirection that renders each component in the sound subsystem independent of the others. The core also plays an important role in exporting the ALSA API to higher applications. The following */dev/snd/** device nodes shown in Figure 13.3 are created and managed by the ALSA core: */dev/snd/controlC0* is a control node (that applications use for controlling volume gain and such), */dev/snd/pcmC0D0p* is a playback device (*p* at the end of the device name stands for playback), and */dev/snd/pcmC0D0c* is a recording device (*c* at the end of the device name stands for capture). In these device names, the integer following *C* is the card number, and that after *D* is the device number. An ALSA driver for a card that has a voice codec for telephony and a stereo codec for music might export */dev/snd/pcmC0D0p* to read audio streams destined for the former and */dev/snd/pcmC0D1p* to channel music bound for the latter.

- Audio controller drivers specific to controller hardware. To drive the audio controller present in the Intel ICH South Bridge chipsets, for example, use the *snd_intel8x0* driver.

- Audio codec interfaces that assist communication between controllers and codecs. For AC'97 codecs, use the *snd_ac97_codec* and *ac97_bus* modules.

- An OSS emulation layer that acts as a conduit between OSS-aware applications and the ALSA-enabled kernel. This layer exports */dev* nodes that mirror what the OSS layer offered in the 2.4 kernels. These nodes, such as */dev/dsp*, */dev/adsp*, and */dev/mixer*, allow OSS applications to run unchanged over ALSA. The OSS */dev/dsp* node maps to the ALSA nodes */dev/snd/pcmC0D0**, */dev/adsp* corresponds to */dev/snd/pcmC0D1**, and */dev/mixer* associates with */dev/snd/controlC0*.

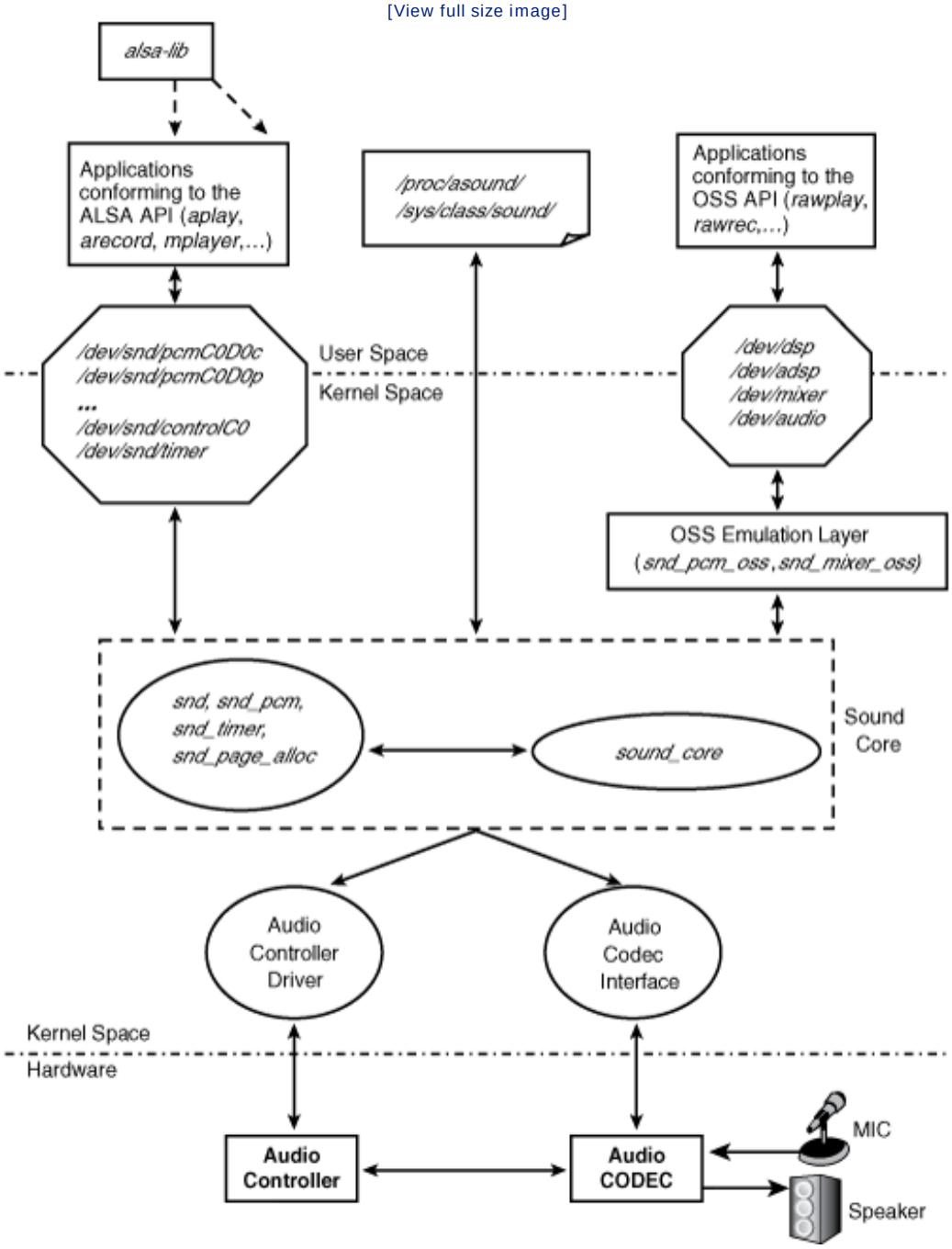
- Procfs and sysfs interface implementations for accessing information via */proc/asound/* and */sys/class/sound/*.

- The user-space ALSA library, *alsa-lib*, which provides the *libasound.so* object. This library eases the job of the ALSA application programmer by offering several canned routines to access ALSA drivers.

- The *alsa-utils* package that includes utilities such as *alsamixer*, *amixer*, *alsactl*, and *aplay*. Use alsamixer or amixer to change volume levels of audio signals such as line-in, line-out, or microphone, and alsactl to control settings for ALSA drivers. To play audio over ALSA, use aplay.

Figure 13.3. Linux-Sound (ALSA) subsystem.

[View full size image]



To obtain a better understanding of the architecture of the Linux-Sound layer, let's look at the ALSA driver modules running on a laptop in tandem with Figure 13.3 (⟶ is used to attach comments):

Code View:

```
bash> lsmod|grep snd
```

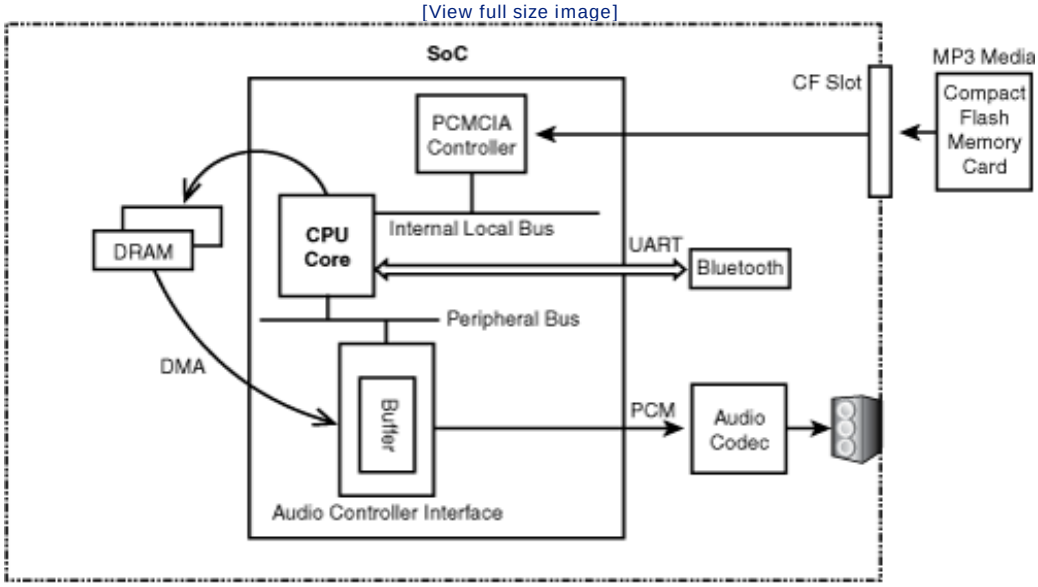
```
snd_intel8x0       33148  0                                  → Audio Controller Driver
snd_ac97_codec     92000  1 snd_intel8x0                     → Audio Codec Interface
ac97_bus           3104   1 snd_ac97_codec                   → Audio Codec Bus
snd_pcm_oss        40512  0                                  → OSS Emulation
snd_mixer_oss      16640  1 snd_pcm_oss                      → OSS Volume Control
snd_pcm            73316  3 snd_intel8x0,snd_ac97_codec,snd_pcm_oss
                                                             → Core layer
snd_timer          22148  1 snd_pcm                          → Core layer
snd                50820  6 snd_intel8x0,snd_ac97_codec,snd_pcm_oss,
                            snd_mixer_oss,snd_pcm,snd_timer
                                                             → Core layer
soundcore          8960   1 snd                              → Core layer
snd_page_alloc     10344  2 snd_intel8x0,snd_pcm             → Core layer
```

Device Example: MP3 Player

Figure 13.4 shows audio operation on an example Linux Bluetooth MP3 player built around an embedded SoC. You can program the Linux cell phone (that we used in Chapter 6, "Serial Drivers," and Chapter 12, "Video Drivers") to download songs from the Internet at night when phone rates are presumably cheaper and upload it to the MP3 player's *Compact Flash* (CF) disk via Bluetooth so that you can listen to the songs next day during office commute.

Figure 13.4. Audio on a Linux MP3 player.

[View full size image]



Our task is to develop the audio software for this device. An application on the player reads songs from the CF disk and decodes it into system memory. A kernel ALSA driver gathers the music data from system memory and dispatches it to transmit buffers that are part of the SoC's audio controller. This PCM data is forwarded to the codec, which plays the music through the device's speaker. As in the case of the navigation system discussed in the preceding chapter, we will assume that Linux supports this SoC, and that all architecture-dependent services such as DMA are supported by the kernel.

The audio software for the MP3 player thus consists of two parts:

1. A user program that decodes MP3 files reads from the CF disk and converts it into raw PCM. To write a native ALSA decoder application, you can leverage the helper routines offered by the *alsa-lib* library. The section "ALSA Programming" looks at how ALSA applications interact with ALSA drivers.

   You also have the option of customizing public domain MP3 players such as *madplay* (http://sourceforge.net/projects/mad/) to suit this device.

2. A low-level kernel ALSA audio driver. The following section is devoted to writing this driver.

One possible hardware implementation of the device shown in Figure 13.4 is by using a PowerPC 405LP SoC and a Texas Instruments TLV320 audio codec. The CPU core in that case is the 405 processor and the on-chip audio controller is the Codec Serial Interface (CSI). SoCs commonly have a high-performance internal local bus that connects to controllers, such as DRAM and video, and a separate on-chip peripheral bus to interface with low-speed peripherals such as serial ports, I^2C, and GPIO. In the case of the 405LP, the former is called the Processor Local Bus (PLB) and the latter is known as the On-chip Peripheral Bus (OPB). The PCMCIA/CF controller hangs off the PLB, whereas the audio controller interface connects to the OPB.

An audio driver is built out of three main ingredients:

1. Routines that handle playback
2. Routines that handle capture
3. Mixer control functions

Our driver implements playback, but does not support recording because the MP3 player in the example has no microphone. The driver also simplifies the mixer function. Rather than offering the full compliment of volume controls, such as speaker, earphone, and line-out, it allows only a single generic volume control.

The register layout of the MP3 player's audio hardware shown in Table 13.1 mirrors these assumptions and simplifications, and does not conform to standards such as AC'97 alluded to earlier. So, the codec has a `SAMPLING_RATE_REGISTER` to configure the playback (digital-to-analog) sampling rate but no registers to set the capture (analog-to-digital) rate. The `VOLUME_REGISTER` configures a single global volume.

Table 13.1. Register Layout of the Audio Hardware in Figure 13.4

Register Name	Description
`VOLUME_REGISTER`	Controls the codec's global volume.
`SAMPLING_RATE_REGISTER`	Sets the codec's sampling rate for digital-to-analog conversion.
`CLOCK_INPUT_REGISTER`	Configures the codec's clock source, divisors, and so on.
`CONTROL_REGISTER`	Enables interrupts, configures interrupt cause (such as completion of a buffer transfer), resets hardware, enables/disables bus operation, and so on.
`STATUS_REGISTER`	Status of codec audio events.
`DMA_ADDRESS_REGISTER`	The example hardware supports a single DMA buffer descriptor. Real-world cards may support multiple descriptors and may have additional registers to hold

Register Name	Description
	parameters such as the descriptor that is currently being processed, the position of the current sample inside the buffer, and so on. DMA is performed to the buffers in the audio controller, so this register resides in the controller's memory space.
DMA_SIZE_REGISTER	Holds the size of the DMA transfer to/from the SoC. This register also resides inside the audio controller.

Listing 13.1 is a skeletal ALSA audio driver for the MP3 player and liberally employs pseudo code (within comments) to cut out extraneous detail. ALSA is a sophisticated framework, and conforming audio drivers are usually several thousand lines long. Listing 13.1 gets you started only on your audio driver explorations. Continue your learning by falling back to the mighty Linux-Sound sources inside the top-level *sound/* directory.

Driver Methods and Structures

Our example driver is implemented as a platform driver. Let's take a look at the steps performed by the platform driver's `probe()` method, `mycard_audio_probe()`. We will digress a bit under each step to explain related concepts and important data structures that we encounter, and this will take us to other parts of the driver and help tie things together.

`mycard_audio_probe()`does the following:

1. Creates an instance of a sound card by invoking `snd_card_new()`:

   ```
   struct snd_card *card = snd_card_new(-1, id[dev->id], THIS_MODULE, 0);
   ```

   The first argument to `snd_card_new()` is the card index (that identifies this card among multiple sound cards in the system), the second argument is the ID that'll be stored in the `id` field of the returned `snd_card` structure, the third argument is the `owner` module, and the last argument is the size of a private data field that'll be made available via the `private_data` field of the returned `snd_card` structure (usually to store chip-specific data such as interrupt levels and I/O addresses).

   `snd_card` represents the created sound card and is defined as follows in *include/sound/core.h*:

   ```
   struct snd_card {
     int number;             /* Card index */
     char id[16];            /* Card ID */
     /* ... */
     struct module *module;  /* Owner module */
     void *private_data;     /* Private data */
     /* ... */
     struct list_head controls;
                             /* All controls for this card */
     struct device *dev;     /* Device assigned to this card*/
     /* ... */
   };
   ```

   The `remove()` counterpart of the probe method, `mycard_audio_remove()`, releases the `snd_card` from the ALSA framework using `snd_card_free()`.

2. Creates a PCM playback instance and associates it with the card created in Step 1, using `snd_pcm_new()`:

```
int snd_pcm_new(struct snd_card *card, char *id,
                int device,
                int playback_count, int capture_count,
                struct snd_pcm **pcm);
```

The arguments are, respectively, the sound card instance created in Step 1, an identifier string, the device index, the number of supported playback streams, the number of supported capture streams (0 in our example), and a pointer to store the allocated PCM instance. The allocated PCM instance is defined as follows in *include/sound/pcm.h*:

Code View:

```
struct snd_pcm {
  struct snd_card *card;          /* Associated snd_card */
  /* ... */
  struct snd_pcm_str streams[2]; /* Playback and capture streams of this PCM
                                    component. Each stream may support
                                    substreams if your h/w supports it
                                  */
  /* ... */
  struct device *dev;             /* Associated hardware
                                    device */
};
```

The `snd_device_new()` routine lies at the core of `snd_pcm_new()` and other similar component instantiation functions. `snd_device_new()` ties a component and a set of operations with the associated `snd_card` (see Step 3).

3. Connects playback operations with the PCM instance created in Step 2, by calling `snd_pcm_set_ops()`. The `snd_pcm_ops` structure specifies these operations for transferring PCM audio to the codec. Listing 13.1 accomplishes this as follows:

Code View:

```
/* Operators for the PCM playback stream */
static struct snd_pcm_ops mycard_playback_ops = {
  .open = mycard_pb_open,         /* Open */
  .close = mycard_pb_close,       /* Close */
  .ioctl = snd_pcm_lib_ioctl,     /* Use to handle special commands, else
                                    specify the generic ioctl handler
                                    snd_pcm_lib_ioctl()*/
  .hw_params = mycard_hw_params, /* Called when higher layers set hardware
                                    parameters such as audio format. DMA
                                    buffer allocation is also done from here */
  .hw_free = mycard_hw_free,      /* Free resources allocated in
                                    mycard_hw_params() */
  .prepare = mycard_pb_prepare,  /* Prepare to transfer the audio stream.
                                    Set audio format such as S16_LE
                                    (explained soon), enable interrupts,.. */
  .trigger = mycard_pb_trigger,  /* Called when the PCM engine starts,
                                    stops, or pauses. The second argument
                                    specifies why it was called. This
```

```
                                        function cannot go to sleep */
};

/* Connect the operations with the PCM instance */
snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &mycard_playback_ops);
```

In Listing 13.1, `mycard_pb_prepare()` configures the sampling rate into the `SAMPLING_RATE_REGISTER`, clock source into the `CLOCKING_INPUT_REGISTER`, and transmit complete interrupt enablement into the `CONTROL_REGISTER`. The `trigger()` method, `mycard_pb_trigger()`, maps an audio buffer populated by the ALSA framework on-the-fly using `dma_map_single()`. (We discussed streaming DMA in Chapter 10, "Peripheral Component Interconnect.") The mapped DMA buffer address is programmed into the `DMA_ADDRESS_REGISTER`. This register is part of the audio controller in the SoC, unlike the earlier registers that reside inside the codec. The audio controller forwards the DMA'ed data to the codec for playback.

Another related object is the `snd_pcm_hardware` structure, which announces the PCM component's hardware capabilities. For our example device, this is defined in Listing 13.1 as follows:

Code View:

```
/* Hardware capabilities of the PCM playback stream */
static struct snd_pcm_hardware mycard_playback_stereo = {
  .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_PAUSE |
           SNDRV_PCM_INFO_RESUME);   /* mmap() is supported. The stream has
                                        pause/resume capabilities */
  .formats = SNDRV_PCM_FMTBIT_S16_LE,/* Signed 16 bits per channel, little
                                        endian */
  .rates = SNDRV_PCM_RATE_8000_48000,/* DAC Sampling rate range */
  .rate_min = 8000,                  /* Minimum sampling rate */
  .rate_max = 48000,                 /* Maximum sampling rate */
  .channels_min = 2,                 /* Supports a left and a right channel */
  .channels_max = 2,                 /* Supports a left and a right channel */
  .buffer_bytes_max = 32768,         /* Max buffer size */
};
```

This object is tied with the associated `snd_pcm` from the `open()` operator, `mycard_playback_open()`, using the PCM *runtime* instance. Each open PCM stream has a runtime object called `snd_pcm_runtime` that contains all information needed to manage that stream. This is a gigantic structure of software and hardware configurations defined in *include/sound/pcm.h* and contains `snd_pcm_hardware` as one of its component fields.

4. Preallocates buffers using `snd_pcm_lib_preallocate_pages_for_all()`. DMA buffers are subsequently obtained from this preallocated area by `mycard_hw_params()` using `snd_pcm_lib_malloc_pages()` and stored in the PCM runtime instance alluded to in Step 3. `mycard_pb_trigger()` DMA-maps this buffer while starting a PCM operation and unmaps it while stopping the PCM operation.

5. Associates a mixer control element with the sound card using `snd_ctl_add()` for global volume control:

   ```
   snd_ctl_add(card, snd_ctl_new1(&mycard_playback_vol, &myctl_private));
   ```

   `snd_ctl_new1()` takes an `snd_kcontrol_new` structure as its first argument and returns a pointer to an `snd_kcontrol` structure. Listing 13.1 defines this as follows:

```
static struct snd_kcontrol_new mycard_playback_vol = {
  .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
                              /* Ctrl element is of type MIXER */
  .name  = "MP3 volume",      /* Name */
  .index = 0,                 /* Codec No: 0 */
  .info  = mycard_pb_vol_info, /* Volume info */
  .get   = mycard_pb_vol_get,  /* Get volume */
  .put   = mycard_pb_vol_put,  /* Set volume */
};
```

The `snd_kcontrol` structure describes a control element. Our driver uses it as a knob for general volume control. `snd_ctl_add()` registers an `snd_kcontrol` element with the ALSA framework. The constituent control methods are invoked when user applications such as *alsamixer* are executed. In Listing 13.1, the `snd_kcontrol put()` method, `mycard_playback_volume_put()`, writes requested volume settings to the codec's `VOLUME_REGISTER`.

6. And finally, registers the sound card with the ALSA framework:

```
snd_card_register(card);
```

`codec_write_reg()` (used, but left unimplemented in Listing 13.1) writes values to codec registers by communicating over the bus that connects the audio controller in the SoC to the external codec. If the underlying bus protocol is I^2C or SPI, for example, `codec_write_reg()` uses the interface functions discussed in Chapter 8, "The Inter-Integrated Circuit Protocol."

If you want to create a */proc* interface in your driver for dumping registers during debug or to export a parameter during normal operation, use the services of `snd_card_proc_new()` and friends. Listing 13.1 does not use */proc* interface files.

If you build and load the driver module in Listing 13.1, you will see two new device nodes appearing on the MP3 player: */dev/snd/pcmC0D0p* and */dev/snd/controlC0*. The former is the interface for audio playback, and the latter is the interface for mixer control. The MP3 decoder application, with the help of *alsa-lib*, streams music by operating over these device nodes.

Listing 13.1. ALSA Driver for the Linux MP3 Player

Code View:

```
include <linux/platform_device.h>
#include <linux/soundcard.h>
#include <sound/driver.h>
#include <sound/core.h>
#include <sound/pcm.h>
#include <sound/initval.h>
#include <sound/control.h>

/* Playback rates supported by the codec */
static unsigned int mycard_rates[] = {
  8000,
  48000,
};
/* Hardware constraints for the playback channel */
static struct snd_pcm_hw_constraint_list mycard_playback_rates = {
  .count = ARRAY_SIZE(mycard_rates),
  .list = mycard_rates,
```

```
  .mask = 0,
};

static struct platform_device *mycard_device;
static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;

/* Hardware capabilities of the PCM stream */
static struct snd_pcm_hardware mycard_playback_stereo = {
  .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_BLOCK_TRANSFER),
  .formats = SNDRV_PCM_FMTBIT_S16_LE, /* 16 bits per channel, little endian */
  .rates = SNDRV_PCM_RATE_8000_48000, /* DAC Sampling rate range */
  .rate_min = 8000,                   /* Minimum sampling rate */
  .rate_max = 48000,                  /* Maximum sampling rate */
  .channels_min = 2,                  /* Supports a left and a right channel */
  .channels_max = 2,                  /* Supports a left and a right channel */
  .buffer_bytes_max = 32768,          /* Maximum buffer size */
};

/* Open the device in playback mode */
static int
mycard_pb_open(struct snd_pcm_substream *substream)
{
  struct snd_pcm_runtime *runtime = substream->runtime;

  /* Initialize driver structures */
  /* ... */
  /* Initialize codec registers */
  /* ... */
  /* Associate the hardware capabilities of this PCM component */
  runtime->hw = mycard_playback_stereo;

  /* Inform the ALSA framework about the constraints that
     the codec has. For example, in this case, it supports
     PCM sampling rates of 8000Hz and 48000Hz only */
  snd_pcm_hw_constraint_list(runtime, 0,
                             SNDRV_PCM_HW_PARAM_RATE,
                             &mycard_playback_rates);
  return 0;
}

/* Close */
static int
mycard_pb_close(struct snd_pcm_substream *substream)
{
  /* Disable the codec, stop DMA, free data structures */
  /* ... */
  return 0;
}

/* Write to codec registers by communicating over
   the bus that connects the SoC to the codec */
void
codec_write_reg(uint codec_register, uint value)
{
  /* ... */
}
/* Prepare to transfer an audio stream to the codec */
static int
mycard_pb_prepare(struct snd_pcm_substream *substream)
```

```
{

  /* Enable Transmit DMA complete interrupt by writing to
     CONTROL_REGISTER using codec_write_reg() */

  /* Set the sampling rate by writing to SAMPLING_RATE_REGISTER */

  /* Configure clock source and enable clocking by writing
     to CLOCK_INPUT_REGISTER */

  /* Allocate DMA descriptors for audio transfer */

  return 0;
}

/* Audio trigger/stop/.. */
static int
mycard_pb_trigger(struct snd_pcm_substream *substream, int cmd)
{
  switch (cmd) {
  case SNDRV_PCM_TRIGGER_START:
    /* Map the audio substream's runtime audio buffer (which is an
       offset into runtime->dma_area) using dma_map_single(),
       populate the resulting address to the audio controller's
       DMA_ADDRESS_REGISTER, and perform DMA */
    /* ... */
    break;

  case SNDRV_PCM_TRIGGER_STOP:
    /* Shut the stream. Unmap DMA buffer using dma_unmap_single() */
    /* ... */
    break;

  default:
    return -EINVAL;
    break;
  }

  return 0;
}
/* Allocate DMA buffers using memory preallocated for DMA from the
   probe() method. dma_[map|unmap]_single() operate on this area
   later on */
static int
mycard_hw_params(struct snd_pcm_substream *substream,
                 struct snd_pcm_hw_params *hw_params)
{
  /* Use preallocated memory from mycard_audio_probe() to
     satisfy this memory request */
  return snd_pcm_lib_malloc_pages(substream,
                                  params_buffer_bytes(hw_params));
}

/* Reverse of mycard_hw_params() */
static int
mycard_hw_free(struct snd_pcm_substream *substream)
{
  return snd_pcm_lib_free_pages(substream);
}
```

```
/* Volume info */
static int
mycard_pb_vol_info(struct snd_kcontrol *kcontrol,
                   struct snd_ctl_elem_info *uinfo)
{
  uinfo->type = SNDRV_CTL_ELEM_TYPE_INTEGER;
                                /* Integer type */
  uinfo->count = 1;             /* Number of values */
  uinfo->value.integer.min = 0;   /* Minimum volume gain */
  uinfo->value.integer.max = 10;  /* Maximum volume gain */
  uinfo->value.integer.step = 1;  /* In steps of 1 */
  return 0;
}

/* Playback volume knob */
static int
mycard_pb_vol_put(struct snd_kcontrol *kcontrol,
                  struct snd_ctl_elem_value *uvalue)
{
  int global_volume = uvalue->value.integer.value[0];

  /* Write global_volume to VOLUME_REGISTER
     using codec_write_reg() */
  /* ... */
  /* If the volume changed from the current value, return 1.
     If there is an error, return negative code. Else return 0 */
}

/* Get playback volume */
static int
mycard_pb_vol_get(struct snd_kcontrol *kcontrol,
                  struct snd_ctl_elem_value *uvalue)
{
  /* Read global_volume from VOLUME_REGISTER
     and return it via uvalue->integer.value[0] */
  /* ... */
  return 0;
}

/* Entry points for the playback mixer */
static struct snd_kcontrol_new mycard_playback_vol = {
  .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
                               /* Control is of type MIXER */
  .name  = "MP3 Volume",       /* Name */
  .index = 0,                  /* Codec No: 0 */
  .info  = mycard_pb_vol_info,  /* Volume info */
  .get   = mycard_pb_vol_get,   /* Get volume */
  .put   = mycard_pb_vol_put,   /* Set volume */
};

/* Operators for the PCM playback stream */
static struct snd_pcm_ops mycard_playback_ops = {
  .open      = mycard_playback_open,    /* Open */
  .close     = mycard_playback_close,   /* Close */
  .ioctl     = snd_pcm_lib_ioctl,       /* Generic ioctl handler */
  .hw_params = mycard_hw_params,        /* Hardware parameters */
  .hw_free   = mycard_hw_free,          /* Free h/w params */
  .prepare   = mycard_playback_prepare, /* Prepare to transfer audio stream */
  .trigger   = mycard_playback_trigger, /* Called when the PCM engine
```

```
                                        starts/stops/pauses */
};

/* Platform driver probe() method */
static int __init
mycard_audio_probe(struct platform_device *dev)
{
  struct snd_card *card;
  struct snd_pcm *pcm;
  int myctl_private;

  /* Instantiate an snd_card structure */
  card = snd_card_new(-1, id[dev->id], THIS_MODULE, 0);

  /* Create a new PCM instance with 1 playback substream
     and 0 capture streams */
  snd_pcm_new(card, "mycard_pcm", 0, 1, 0, &pcm);

  /* Set up our initial DMA buffers */
  snd_pcm_lib_preallocate_pages_for_all(pcm,
                        SNDRV_DMA_TYPE_CONTINUOUS,
                        snd_dma_continuous_data
                        (GFP_KERNEL), 256*1024,
                        256*1024);

  /* Connect playback operations with the PCM instance */
  snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                  &mycard_playback_ops);

  /* Associate a mixer control element with this card */
  snd_ctl_add(card, snd_ctl_new1(&mycard_playback_vol,
                                 &myctl_private));

  strcpy(card->driver, "mycard");

  /* Register the sound card */
  snd_card_register(card);

  /* Store card for access from other methods */
  platform_set_drvdata(dev, card);

  return 0;
}

/* Platform driver remove() method */
static int
mycard_audio_remove(struct platform_device *dev)
{
  snd_card_free(platform_get_drvdata(dev));
  platform_set_drvdata(dev, NULL);
  return 0;
}

/* Platform driver definition */
static struct platform_driver mycard_audio_driver = {
  .probe = mycard_audio_probe,     /* Probe method */
  .remove = mycard_audio_remove,   /* Remove method */
  .driver = {
    .name = "mycard_ALSA",
```

```
  },
};

/* Driver Initialization */
static int __init
mycard_audio_init(void)
{
  /* Register the platform driver and device */
  platform_driver_register(&mycard_audio_driver);

  mycard_device = platform_device_register_simple("mycard_ALSA",
                                                  -1, NULL, 0);
  return 0;
}

/* Driver Exit */
static void __exit
mycard_audio_exit(void)
{
  platform_device_unregister(mycard_device);
  platform_driver_unregister(&mycard_audio_driver);
}

module_init(mycard_audio_init);
module_exit(mycard_audio_exit);
MODULE_LICENSE("GPL");
```

ALSA Programming

To understand how the user space alsa-lib library interacts with kernel space ALSA drivers, let's write a simple application that sets the volume gain of the MP3 player. We will map the alsa-lib services used by the application to the mixer control methods defined in Listing 13.1. Let's begin by loading the driver and examining the mixer's capabilities:

```
bash> amixer contents
...
numid=3,iface=MIXER,name="MP3 Volume"
     ; type=INTEGER,...
...
```

In the volume-control application, first allocate space for the alsa-lib objects necessary to perform the volume-control operation:

```
#include <alsa/asoundlib.h>
snd_ctl_elem_value_t *nav_control;
snd_ctl_elem_id_t    *nav_id;
snd_ctl_elem_info_t  *nav_info;

snd_ctl_elem_value_alloca(&nav_control);
snd_ctl_elem_id_alloca(&nav_id);
snd_ctl_elem_info_alloca(&nav_info);
```

Next, set the interface type to SND_CTL_ELEM_IFACE_MIXER as specified in the mycard_playback_vol structure in Listing 13.1:

```
snd_ctl_elem_id_set_interface(nav_id, SND_CTL_ELEM_IFACE_MIXER);
```

Now set the numid for the MP3 volume obtained from the amixer output above:

```
snd_ctl_elem_id_set_numid(nav_id, 3); /* num_id=3 */
```

Open the mixer node, */dev/snd/controlC0*. The third argument to snd_ctl_open() specifies the card number in the node name:

```
snd_ctl_open(&nav_handle, card, 0);
/* Connect data structures */
snd_ctl_elem_info_set_id(nav_info, nav_id);
snd_ctl_elem_info(nav_handle, nav_info);
```

Elicit the type field in the snd_ctl_elem_info structure defined in mycard_pb_vol_info() in Listing 13.1 as follows:

```
if (snd_ctl_elem_info_get_type(nav_info) !=
                      SND_CTL_ELEM_TYPE_INTEGER) {
  printk("Mismatch in control type\n");
}
```

Get the supported codec volume range by communicating with the mycard_pb_vol_info() driver method:

```
long desired_volume = 5;
long min_volume = snd_ctl_elem_info_get_min(nav_info);
long max_volume = snd_ctl_elem_info_get_max(nav_info);
/* Ensure that the desired_volume is within min_volume and
   max_volume */
/* ... */
```

As per the definition of mycard_pb_vol_info() in Listing 13.1, the minimum and maximum values returned by the above alsa-lib helper routines are 0 and 10, respectively.

Finally, set the desired volume and write it to the codec:

```
snd_ctl_elem_value_set_integer(nav_control, 0, desired_volume);
snd_ctl_elem_write(nav_handle, nav_control);
```

The call to snd_ctl_elem_write() results in the invocation of mycard_pb_vol_put(), which writes the desired volume gain to the codec's VOLUME_REGISTER.

MP3 Decoding Complexity

The MP3 decoder application running on the player, as shown in Figure 13.4, requires a supply rate of MP3 frames from the CF disk that can sustain the common MP3 sampling rate of 128KBps. This is usually not a problem for most low-MIPs devices, but in case it is, consider buffering each song in memory before decoding it. (MP3 frames at 128KBps roughly consume 1MB per minute of music.)

MP3 decoding is lightweight and can usually be accomplished on-the-fly, but MP3 encoding is heavy-duty and cannot be achieved in real time without hardware assist. Voice codecs such as G.711 and G.729 used in Voice over IP (VoIP) environments can, however, encode and decode audio data in real time.

Debugging

You may turn on options under *Device Drivers* → *Sound* → *Advanced Linux Sound Architecture* in the kernel configuration menu to include ALSA debug code (`CONFIG_SND_DEBUG`), verbose `printk()` messages (`CONFIG_SND_VERBOSE_PRINTK`), and verbose procfs content (`CONFIG_SND_VERBOSE_PROCFS`).

Procfs information pertaining to ALSA drivers resides in */proc/asound/*. Look inside */sys/class/sound/* for the device model information associated with each sound-class device.

If you think you have found a bug in an ALSA driver, post it to the alsa-devel mailing list (http://mailman.alsa-project.org/mailman/listinfo/alsa-devel). The linux-audio-dev mailing list (http://music.columbia.edu/mailman/listinfo/linux-audio-dev/), also called the *Linux Audio Developers* (LAD) list, discusses questions related to the Linux-sound architecture and audio applications.

Looking at the Sources

The sound core, audio buses, architectures, and the obsolete OSS suite all have their own separate subdirectories under *sound/*. For the AC'97 interface implementation, look inside *sound/pci/ac97/*. For an example I^2S-based audio driver, look at *sound/soc/at91/at91-ssc.c*, the audio driver for Atmel's AT91-series ARM-based embedded SoCs. Use *sound/drivers/dummy.c* as a starting point for developing your custom ALSA driver if you cannot find a closer match.

*Documentation/sound/** contains information on ALSA and OSS drivers. *Documentation/sound/alsa/DocBook/* contains a DocBook on writing ALSA drivers. An ALSA configuration guide is available in *Documentation/sound/alsa/ALSA-Configuration.txt*. The Sound-HOWTO, downloadable from http://tldp.org/HOWTO/Sound-HOWTO/, answers several frequently asked questions pertaining to Linux support for audio devices.

Madplay is a software MP3 decoder and player that is both ALSA- and OSS-aware. You can look at its sources for tips on user-space audio programming.

Two no-frills OSS tools for basic playback and recording are *rawplay* and *rawrec*, whose sources are downloadable from http://rawrec.sourceforge.net/.

You can find the home page of the Linux-ALSA project at www.alsa-project.org. Here, you will find the latest news on ALSA drivers, details on the ALSA programming API, and information on subscribing to related mailing lists. Sources of alsa-utils and alsa-lib, downloadable from this page, can aid you while developing ALSA-aware applications.

Table 13.2 contains the main data structures used in this chapter and their location in the source tree. Table 13.3 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 13.2. Summary of Data Structures

Data Structure	Location	Description
`snd_card`	*include/sound/core.h*	Representation of a sound card
`snd_pcm`	*include/sound/pcm.h*	An instance of a PCM object
`snd_pcm_ops`	*include/sound/pcm.h*	Used to connect operations with a PCM object
`snd_pcm_substream`	*include/sound/pcm.h*	Information about the current audio stream
`snd_pcm_runtime`	*include/sound/pcm.h*	Runtime details of the audio stream
`snd_kcontrol_new`	*include/sound/control.h*	Representation of an ALSA control element

Table 13.3. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`snd_card_new()`	*sound/core/init.c*	Instantiates an `snd_card` structure
`snd_card_free()`	*sound/core/init.c*	Frees an instantiated `snd_card`

Kernel Interface	Location	Description
`snd_card_register()`	*sound/core/init.c*	Registers a sound card with the ALSA framework
`snd_pcm_lib_preallocate_pages_for_all()`	*sound/core/pcm_memory.c*	Preallocates buffers for a sound card
`snd_pcm_lib_malloc_pages()`	*sound/core/pcm_memory.c*	Allocates DMA buffers for a sound card
`snd_pcm_new()`	*sound/core/pcm.c*	Creates an instance of a PCM object
`snd_pcm_set_ops()`	*sound/core/pcm_lib.c*	Connects playback or capture operations with a PCM object
`snd_ctl_add()`	*sound/core/control.c*	Associates a mixer control element with a sound card
`snd_ctl_new1()`	*sound/core/control.c*	Allocates an `snd_kcontrol` structure and initializes it with supplied control operations
`snd_card_proc_new()`	*sound/core/info.c*	Creates a */proc* entry and assigns it to a card instance

Chapter 14. Block Drivers

In This Chapter

- Storage Technologies 416
- Linux Block I/O Layer 421
- I/O Schedulers 422
- Block Driver Data Structures and Methods 423
- Device Example: Simple Storage Controller 426
- Advanced Topics 434
- Debugging 436
- Looking at the Sources 437

Block devices are storage media capable of random access. Unlike character devices, block devices can hold filesystem data. In this chapter, let's find out how Linux supports storage buses and devices.

Storage Technologies

Let's start by taking a tour of the popular storage technologies found in today's computer systems. We'll also associate these technologies with the corresponding device driver subsystems in the kernel source tree.

Integrated Drive Electronics (IDE) is the common storage interface technology used in the PC environment. ATA (short for *Advanced Technology Attachment*) is the official name for the related specifications. The IDE/ATA

standard began with ATA-1; the latest version is ATA-7 and supports bandwidths of up to 133MBps. Intervening versions of the specification are ATA-2, which introduced *logical block addressing* (LBA); ATA-3, which enabled SMART-capable disks (discussed later); ATA-4, which brought support for Ultra DMA and the associated 33MBps throughput; ATA-5, which increased maximum transfer speeds to 66MBps; and ATA-6, which provided for 100MBps data rates.

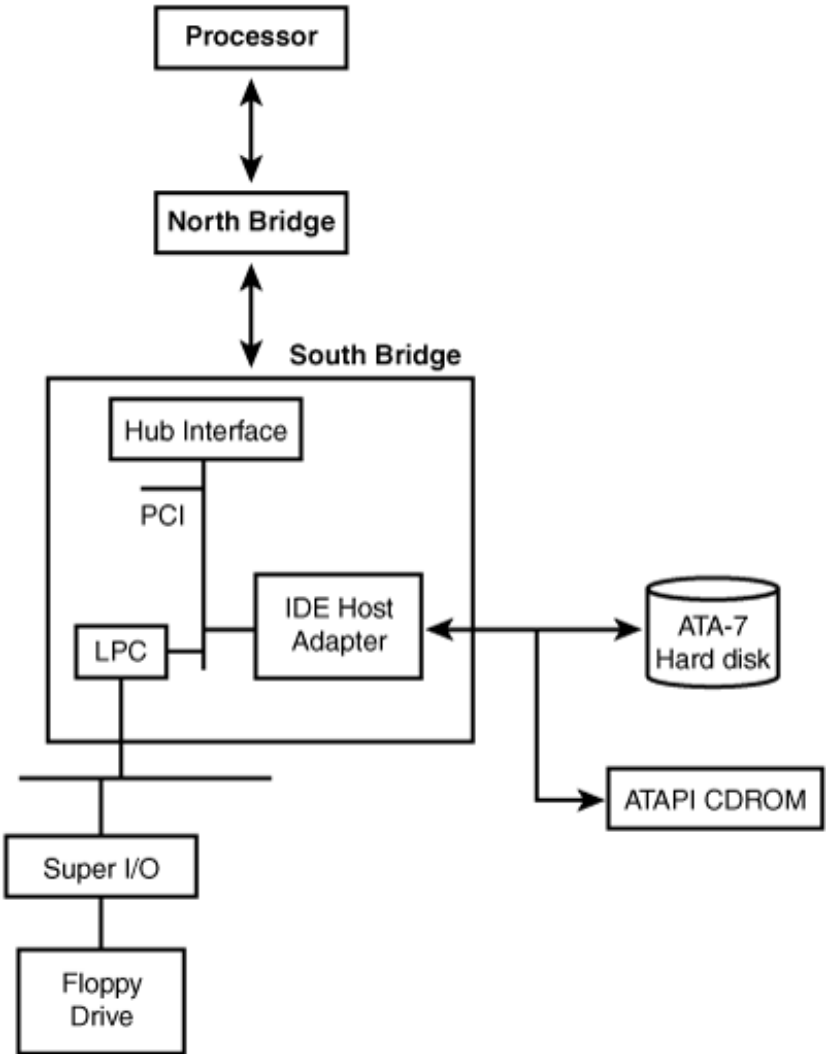
Storage devices such as CD-ROMs and tapes connect to the standard IDE cable using a special protocol called the *ATA Packet Interface* (ATAPI).[1]ATAPI was introduced along with ATA-4.

[1] The ATAPI protocol is closer to SCSI than to IDE.

The floppy disk controller in PC systems has traditionally been part of the Super I/O chipset about which we learned in Chapter 6, "Serial Drivers." These internal drives, however, have given way to faster external USB floppy drives in today's PC environment.

Figure 14.1 shows an ATA-7 disk drive connected to an IDE host adapter that's part of the South Bridge chipset on a PC system. Also shown connected are an ATAPI CD-ROM drive and a floppy drive.

Figure 14.1. Storage media in a PC system.



IDE/ATA is a parallel bus technology (sometimes called *Parallel ATA* or PATA) and cannot scale to high speeds, as you learned while discussing PCIe in Chapter 10, "Peripheral Component Interconnect." *Serial ATA* (SATA) is a modern serial bus evolution of PATA that supports transfer speeds in the realm of 300MBps and beyond. In addition to offering higher throughput than PATA, SATA brings capabilities such as hot swapping. SATA technology is steadily replacing PATA. See the sidebar "libATA" to learn about the new ATA subsystem in the kernel that supports both SATA and PATA.

libATA

libATA is the new ATA subsystem in the Linux kernel. It consists of a set of ATA library routines and a collection of low-level drivers that use them. libATA supports both SATA and PATA. SATA drivers in libATA have been around for some time under *drivers/scsi/*, but PATA drivers and the new *drivers/ata/* directory that now houses all libATA sources were introduced with the 2.6.19 kernel release.

If your system is enabled with SATA storage, you need the services of libATA in tandem with the SCSI subsystem. libATA support for PATA is still experimental, and by default, PATA drivers continue to use the legacy IDE drivers that live in *drivers/ide/*.

Assume that your system is SATA-enabled via an Intel ICH7 South Bridge chipset. You need the following libATA components to access your disk:

1. The libATA core— To enable this, set `CONFIG_ATA` during kernel configuration. For a list of library functions offered by the core, grep for `EXPORT_SYMBOL_GPL` inside the *drivers/ata/* directory.

2. Advanced Host Controller Interface (AHCI) support— AHCI specifies the register interface supported by SATA host adapters and is enabled by choosing `CONFIG_AHCI` at configuration time.

3. The host controller adapter driver— For the ICH7, enable `CONFIG_ATA_PIIX`.

Additionally, you need the mid-level and upper-level SCSI drivers (`CONFIG_SCSI` and friends). After you have loaded all these kernel components, your SATA disk partitions appear to the system as */dev/sd**, just like SCSI or USB mass storage partitions.

The home page of the libATA project is http://linux-ata.org/. A DocBook is available as part of the kernel source tree in *Documentation/DocBook/libata.tmpl*. A libATA developer's guide is available at www.kernel.org/pub/linux/kernel/people/jgarzik/libata.pdf.

Small Computer System Interface (SCSI) is the storage technology of choice in servers and high-end workstations. SCSI is somewhat faster than SATA and supports speeds of the order of 320MBps. SCSI has traditionally been a parallel interface standard, but, like ATA, has recently shifted to serial operation with the advent of a bus technology called *Serial Attached SCSI* (SAS).

The kernel's SCSI subsystem is architected into three layers: top-level drivers for media such as disks, CD-ROMs, and tapes; a middle-level layer that scans the SCSI bus and configures devices; and low-level host adapter drivers. We learned about these layers in the section "Mass Storage" in Chapter 11, "Universal Serial Bus." Refer back to Figure 11.4 in that chapter to see how the different components of the SCSI subsystem interact with each other.[2] USB mass storage drives use flash memory internally but communicate with host systems using the SCSI protocol.

[2] SCSI support is discussed in other parts of this book, too. The section "User Mode SCSI" in Chapter 19, "Drivers in User Space," discusses the *SCSI Generic* (sg) interface that lets you directly dispatch commands from user space to SCSI devices. The section "iSCSI" in Chapter 20, "More Devices and Drivers," briefly looks at the iSCSI protocol, which allows the transport of SCSI packets to a remote block device over a TCP/IP network.

Redundant array of inexpensive disks (RAID) is a technology built in to some SCSI and SATA controllers to achieve redundancy and reliability. Various RAID levels have been defined. RAID-1, for example, specifies *disk mirroring*, where data is duplicated on separate disks. Linux drivers are available for several RAID-capable disk drives. The kernel also offers a multidisk (md) driver that implements most RAID levels in software.

Miniature storage is the name of the game in the embedded consumer electronics space. Transfer speeds in this domain are much lower than that offered by the technologies discussed thus far. *Secure Digital* (SD) cards and their smaller form-factor derivatives (miniSD and microSD) are popular storage media[3] in devices such as cameras, cell phones, and music players. Cards complying with version 1.01 of the SD card specification support transfer speeds of up to 10MBps. SD storage has evolved from an older, slower, but compatible technology called *MultiMediaCard* (MMC) that supports data rates of 2.5MBps. The kernel contains an SD/MMC subsystem in *drivers/mmc/*.

[3] See the sidebar "WiFi over SDIO" in Chapter 16, "Linux Without Wires," to learn about nonstorage technologies available in SD form factor.

The section "PCMCIA Storage" in Chapter 9, "PCMCIA and Compact Flash," looked at different PCMCIA/CF flavors of storage cards and their corresponding kernel drivers. PCMCIA memory cards such as microdrives support true IDE operation, whereas those that internally use solid-state memory emulate IDE and export an IDE programming model to the kernel. In both these cases, the kernel's IDE subsystem can be used to enable the card.

Table 14.1 summarizes important storage technologies and the location of the associated device drivers in the kernel source tree.

Table 14.1. Storage Technologies and Associated Device Drivers

Storage Technology	Description	Source File
IDE/ATA	Storage interface technology in the PC environment. Supports data rates of 133MBps for ATA-7.	*drivers/ide/ide-disk.c*, *driver/ide/ide-io.c*, *drivers/ide/ide-probe.c* or *drivers/ata/* (Experimental)
ATAPI	Storage devices such as CD-ROMs and tapes connect to the standard IDE cable using the ATAPI protocol.	*drivers/ide/ide-cd.c* or *drivers/ata/* (Experimental)
Floppy (internal)	The floppy controller resides in the Super I/O chip on the LPC bus in PC-compatible systems. Supports transfer rates of the order of 150KBps.	*drivers/block/floppy.c*
SATA	Serial evolution of IDE/ATA. Supports speeds of 300MBps and beyond.	*drivers/ata/*, *drivers/scsi/*
SCSI	Storage technology popular in the server environment. Supports transfer rates of 320MBps for Ultra320 SCSI.	*drivers/scsi/*

Storage Technology	Description	Source File
USB Mass Storage	This refers to USB hard disks, pen drives, CD-ROMs, and floppy drives. Look at the section "Mass Storage" in Chapter 11. USB 2.0 devices can communicate at speeds of up to 60MBps.	*drivers/usb/storage/*, *drivers/scsi/*
RAID:		
Hardware RAID	This is a capability built into high-end SCSI/SATA disk controllers to achieve redundancy and reliability.	*drivers/scsi/*, *drivers/ata/*
Software RAID	On Linux, the multidisk (md) driver implements several RAID levels in software.	*drivers/md/*
SD/miniSD/microSD	Small form-factor storage media popular in consumer electronic devices such as cameras and cell phones. Supports transfer rates of up to 10MBps.	*drivers/mmc/*
MMC	Older removable storage standard that's compatible with SD cards. Supports data rates of 2.5MBps.	*drivers/mmc/*
PCMCIA/ CF storage cards	PCMCIA/CF form factor of miniature IDE drives, or solid-state memory cards that emulate IDE. See the section "PCMCIA Storage" in Chapter 9.	*drivers/ide/legacy/ide-cs.c* or *drivers/ata/pata_pcmcia.c* (experimental)
Block device emulation over flash memory	Emulates a hard disk over flash memory. See the section "Block Device Emulation" in Chapter 17, "Memory Technology Devices."	*drivers/mtd/mtdblock.c*, *drivers/mtd/mtd_blkdevs.c*
Virtual block devices on Linux:		
RAM disk	Implements support to use a RAM region as a block device.	*drivers/block/rd.c*
Loopback device	Implements support to use a regular file as a block device.	*drivers/block/loop.c*

Chapter 14. Block Drivers

In This Chapter

- Storage Technologies 416
- Linux Block I/O Layer 421
- I/O Schedulers 422
- Block Driver Data Structures and Methods 423
- Device Example: Simple Storage Controller 426
- Advanced Topics 434
- Debugging 436
- Looking at the Sources 437

Block devices are storage media capable of random access. Unlike character devices, block devices can hold filesystem data. In this chapter, let's find out how Linux supports storage buses and devices.

Storage Technologies

Let's start by taking a tour of the popular storage technologies found in today's computer systems. We'll also associate these technologies with the corresponding device driver subsystems in the kernel source tree.

Integrated Drive Electronics (IDE) is the common storage interface technology used in the PC environment. ATA (short for *Advanced Technology Attachment*) is the official name for the related specifications. The IDE/ATA

standard began with ATA-1; the latest version is ATA-7 and supports bandwidths of up to 133MBps. Intervening versions of the specification are ATA-2, which introduced *logical block addressing* (LBA); ATA-3, which enabled SMART-capable disks (discussed later); ATA-4, which brought support for Ultra DMA and the associated 33MBps throughput; ATA-5, which increased maximum transfer speeds to 66MBps; and ATA-6, which provided for 100MBps data rates.

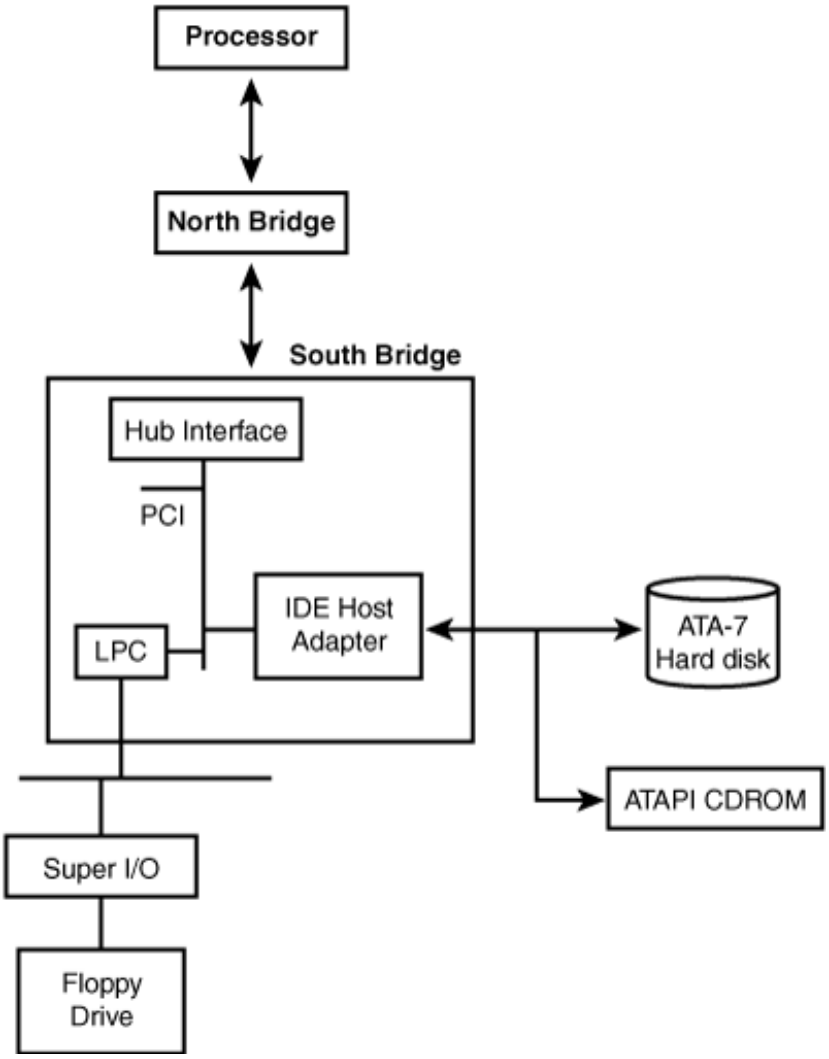
Storage devices such as CD-ROMs and tapes connect to the standard IDE cable using a special protocol called the *ATA Packet Interface* (ATAPI).[1]ATAPI was introduced along with ATA-4.

[1] The ATAPI protocol is closer to SCSI than to IDE.

The floppy disk controller in PC systems has traditionally been part of the Super I/O chipset about which we learned in Chapter 6, "Serial Drivers." These internal drives, however, have given way to faster external USB floppy drives in today's PC environment.

Figure 14.1 shows an ATA-7 disk drive connected to an IDE host adapter that's part of the South Bridge chipset on a PC system. Also shown connected are an ATAPI CD-ROM drive and a floppy drive.

Figure 14.1. Storage media in a PC system.



IDE/ATA is a parallel bus technology (sometimes called *Parallel ATA* or PATA) and cannot scale to high speeds, as you learned while discussing PCIe in Chapter 10, "Peripheral Component Interconnect." *Serial ATA* (SATA) is a modern serial bus evolution of PATA that supports transfer speeds in the realm of 300MBps and beyond. In addition to offering higher throughput than PATA, SATA brings capabilities such as hot swapping. SATA technology is steadily replacing PATA. See the sidebar "libATA" to learn about the new ATA subsystem in the kernel that supports both SATA and PATA.

libATA

libATA is the new ATA subsystem in the Linux kernel. It consists of a set of ATA library routines and a collection of low-level drivers that use them. libATA supports both SATA and PATA. SATA drivers in libATA have been around for some time under *drivers/scsi/*, but PATA drivers and the new *drivers/ata/* directory that now houses all libATA sources were introduced with the 2.6.19 kernel release.

If your system is enabled with SATA storage, you need the services of libATA in tandem with the SCSI subsystem. libATA support for PATA is still experimental, and by default, PATA drivers continue to use the legacy IDE drivers that live in *drivers/ide/*.

Assume that your system is SATA-enabled via an Intel ICH7 South Bridge chipset. You need the following libATA components to access your disk:

1. The libATA core— To enable this, set `CONFIG_ATA` during kernel configuration. For a list of library functions offered by the core, grep for `EXPORT_SYMBOL_GPL` inside the *drivers/ata/* directory.

2. Advanced Host Controller Interface (AHCI) support— AHCI specifies the register interface supported by SATA host adapters and is enabled by choosing `CONFIG_AHCI` at configuration time.

3. The host controller adapter driver— For the ICH7, enable `CONFIG_ATA_PIIX`.

Additionally, you need the mid-level and upper-level SCSI drivers (`CONFIG_SCSI` and friends). After you have loaded all these kernel components, your SATA disk partitions appear to the system as */dev/sd**, just like SCSI or USB mass storage partitions.

The home page of the libATA project is http://linux-ata.org/. A DocBook is available as part of the kernel source tree in *Documentation/DocBook/libata.tmpl*. A libATA developer's guide is available at www.kernel.org/pub/linux/kernel/people/jgarzik/libata.pdf.

Small Computer System Interface (SCSI) is the storage technology of choice in servers and high-end workstations. SCSI is somewhat faster than SATA and supports speeds of the order of 320MBps. SCSI has traditionally been a parallel interface standard, but, like ATA, has recently shifted to serial operation with the advent of a bus technology called *Serial Attached SCSI* (SAS).

The kernel's SCSI subsystem is architected into three layers: top-level drivers for media such as disks, CD-ROMs, and tapes; a middle-level layer that scans the SCSI bus and configures devices; and low-level host adapter drivers. We learned about these layers in the section "Mass Storage" in Chapter 11, "Universal Serial Bus." Refer back to Figure 11.4 in that chapter to see how the different components of the SCSI subsystem interact with each other.[2] USB mass storage drives use flash memory internally but communicate with host systems using the SCSI protocol.

[2] SCSI support is discussed in other parts of this book, too. The section "User Mode SCSI" in Chapter 19, "Drivers in User Space," discusses the *SCSI Generic* (sg) interface that lets you directly dispatch commands from user space to SCSI devices. The section "iSCSI" in Chapter 20, "More Devices and Drivers," briefly looks at the iSCSI protocol, which allows the transport of SCSI packets to a remote block device over a TCP/IP network.

Redundant array of inexpensive disks (RAID) is a technology built in to some SCSI and SATA controllers to achieve redundancy and reliability. Various RAID levels have been defined. RAID-1, for example, specifies *disk mirroring*, where data is duplicated on separate disks. Linux drivers are available for several RAID-capable disk drives. The kernel also offers a multidisk (md) driver that implements most RAID levels in software.

Miniature storage is the name of the game in the embedded consumer electronics space. Transfer speeds in this domain are much lower than that offered by the technologies discussed thus far. *Secure Digital* (SD) cards and their smaller form-factor derivatives (miniSD and microSD) are popular storage media[3] in devices such as cameras, cell phones, and music players. Cards complying with version 1.01 of the SD card specification support transfer speeds of up to 10MBps. SD storage has evolved from an older, slower, but compatible technology called *MultiMediaCard* (MMC) that supports data rates of 2.5MBps. The kernel contains an SD/MMC subsystem in *drivers/mmc/*.

[3] See the sidebar "WiFi over SDIO" in Chapter 16, "Linux Without Wires," to learn about nonstorage technologies available in SD form factor.

The section "PCMCIA Storage" in Chapter 9, "PCMCIA and Compact Flash," looked at different PCMCIA/CF flavors of storage cards and their corresponding kernel drivers. PCMCIA memory cards such as microdrives support true IDE operation, whereas those that internally use solid-state memory emulate IDE and export an IDE programming model to the kernel. In both these cases, the kernel's IDE subsystem can be used to enable the card.

Table 14.1 summarizes important storage technologies and the location of the associated device drivers in the kernel source tree.

Table 14.1. Storage Technologies and Associated Device Drivers

Storage Technology	Description	Source File
IDE/ATA	Storage interface technology in the PC environment. Supports data rates of 133MBps for ATA-7.	*drivers/ide/ide-disk.c*, *driver/ide/ide-io.c*, *drivers/ide/ide-probe.c* or *drivers/ata/* (Experimental)
ATAPI	Storage devices such as CD-ROMs and tapes connect to the standard IDE cable using the ATAPI protocol.	*drivers/ide/ide-cd.c* or *drivers/ata/* (Experimental)
Floppy (internal)	The floppy controller resides in the Super I/O chip on the LPC bus in PC-compatible systems. Supports transfer rates of the order of 150KBps.	*drivers/block/floppy.c*
SATA	Serial evolution of IDE/ATA. Supports speeds of 300MBps and beyond.	*drivers/ata/*, *drivers/scsi/*
SCSI	Storage technology popular in the server environment. Supports transfer rates of 320MBps for Ultra320 SCSI.	*drivers/scsi/*

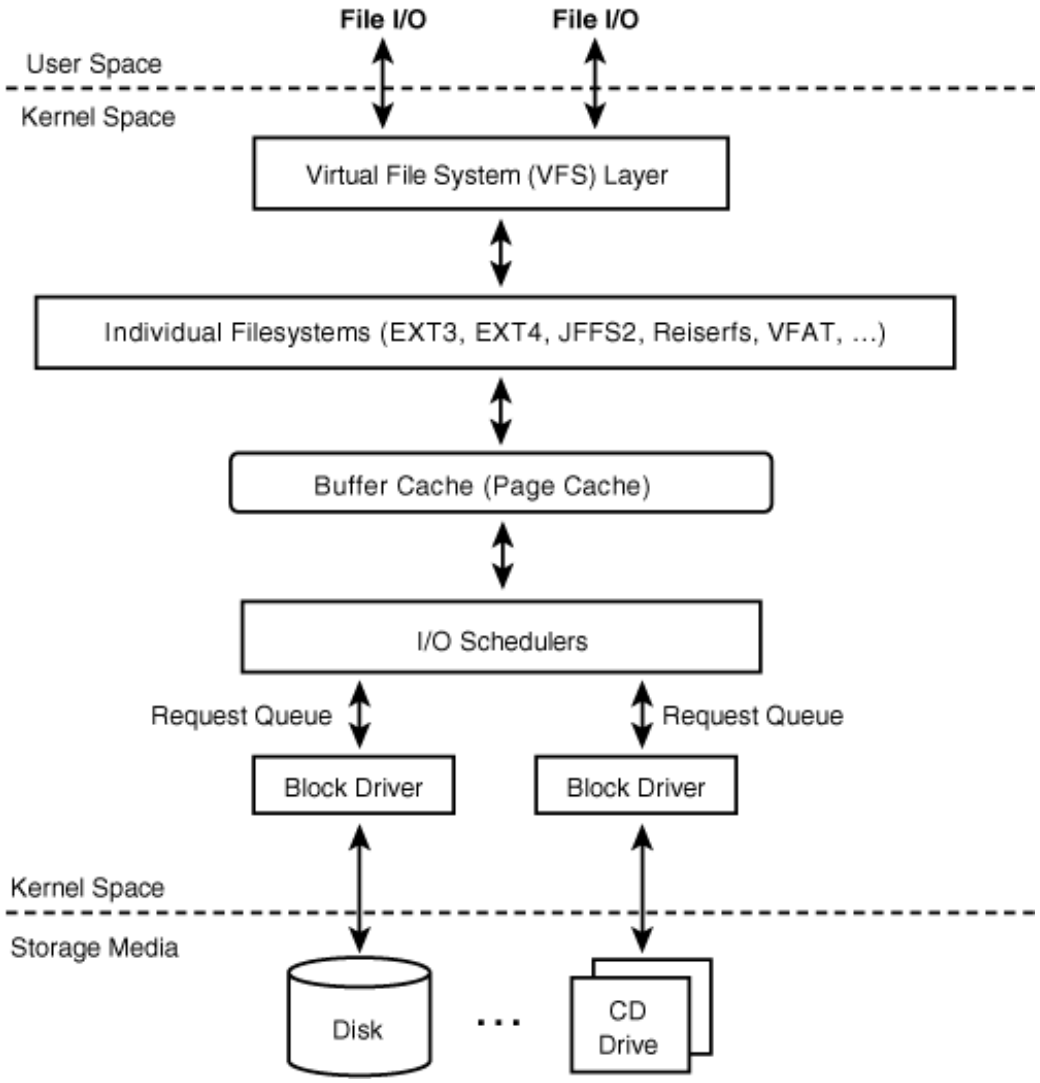
Storage Technology	Description	Source File
USB Mass Storage	This refers to USB hard disks, pen drives, CD-ROMs, and floppy drives. Look at the section "Mass Storage" in Chapter 11. USB 2.0 devices can communicate at speeds of up to 60MBps.	*drivers/usb/storage/*, *drivers/scsi/*
RAID:		
Hardware RAID	This is a capability built into high-end SCSI/SATA disk controllers to achieve redundancy and reliability.	*drivers/scsi/*, *drivers/ata/*
Software RAID	On Linux, the multidisk (md) driver implements several RAID levels in software.	*drivers/md/*
SD/miniSD/microSD	Small form-factor storage media popular in consumer electronic devices such as cameras and cell phones. Supports transfer rates of up to 10MBps.	*drivers/mmc/*
MMC	Older removable storage standard that's compatible with SD cards. Supports data rates of 2.5MBps.	*drivers/mmc/*
PCMCIA/ CF storage cards	PCMCIA/CF form factor of miniature IDE drives, or solid-state memory cards that emulate IDE. See the section "PCMCIA Storage" in Chapter 9.	*drivers/ide/legacy/ide-cs.c* or *drivers/ata/pata_pcmcia.c* (experimental)
Block device emulation over flash memory	Emulates a hard disk over flash memory. See the section "Block Device Emulation" in Chapter 17, "Memory Technology Devices."	*drivers/mtd/mtdblock.c*, *drivers/mtd/mtd_blkdevs.c*
Virtual block devices on Linux:		
RAM disk	Implements support to use a RAM region as a block device.	*drivers/block/rd.c*
Loopback device	Implements support to use a regular file as a block device.	*drivers/block/loop.c*

Linux Block I/O Layer

The block I/O layer was considerably overhauled between the 2.4 and 2.6 kernel releases. The motivation for the redesign was that the block layer, more than other kernel subsystems, has the potential to impact overall system performance.

Let's take a look at Figure 14.2 to learn the workings of the Linux block I/O layer. The storage media contains files residing in a filesystem, such as EXT3 or Reiserfs. User applications invoke I/O system calls to access these files. The resulting filesystem operations pass through the generic *Virtual File System* (VFS) layer before entering the individual filesystem driver. The buffer cache speeds up filesystem access to block devices by caching disk blocks. If a block is found in the buffer cache, the time required to access the disk to read the block is saved. Data destined for each block device is lined up in a request queue. The filesystem driver populates the request queue belonging to the desired block device, whereas the block driver receives and consumes requests from the corresponding queue. In between, *I/O schedulers* manipulate the request queue so as to minimize disk access latencies and maximize throughput.

Figure 14.2. Block I/O on Linux.



Let's next examine the different I/O schedulers available on Linux.

I/O Schedulers

Block devices suffer *seek times*, the latency to move the disk head from its existing position to the disk sector of interest. The main goal of an I/O scheduler is to increase system throughput by minimizing these seek times. To achieve this, I/O schedulers maintain the request queue in sorted order according to the disk sectors associated with the constituent requests. New requests are inserted into the queue such that this order is maintained. If an existing request in the queue is associated with an adjacent disk sector, the new request is merged with it. Because of these properties, I/O schedulers bear an operational resemblance to elevators—they schedule requests in a single direction until the last requester in the line is serviced.

The I/O scheduler in 2.4 kernels implemented a straightforward version of this algorithm and was called the *Linus elevator.* This turned out to be inadequate under real-world conditions, however, and was replaced in the 2.6 kernel by a suite of four schedulers: *Deadline*, *Anticipatory*, *Complete Fair Queuing*, and *Noop.* The scheduler used by default is Anticipatory, but this can be changed during kernel configuration or by changing the value of */sys/block/[disk]/queue/scheduler*. (Replace *[disk]* with *hda* if you are using an IDE disk, for example.) Table 14.2 briefly describes Linux I/O schedulers.

Table 14.2. Linux I/O Schedulers

I/O Scheduler	Description	Source File
Linus elevator	Straightforward implementation of the standard merge-and-sort I/O scheduling algorithm.	*drivers/block/elevator.c* (in the 2.4 kernel tree)
Deadline	In addition to what the Linus elevator does, the Deadline scheduler associates expiration times with each request in order to ensure that a burst of requests to the same disk region do not starve requests to regions that are farther away. Moreover, read operations are granted more priority than write operations because user processes usually block until their read requests complete. The Deadline scheduler thus ensures that each I/O request is serviced within a time limit, which is important for some database loads.	*block/deadline-iosched.c* (in the 2.6 kernel tree)
Anticipatory	Similar to the Deadline scheduler, except that after servicing read requests, the Anticipatory scheduler waits for a predetermined amount of time anticipating further requests. This scheduling technique is suited for workstation/interactive loads.	*block/as-iosched.c* (in the 2.6 kernel tree)
Complete Fair Queuing (CFQ)	Similar to the Linus elevator, except that the CFQ scheduler maintains one request queue per originating process, rather than one generic queue. This ensures that each process (or process group) gets a fair portion of the I/O and prevents one process from starving others.	*block/cfq-iosched.c* (in the 2.6 kernel tree)
Noop	The Noop scheduler doesn't spend time	*block/noop-iosched.c*

I/O Scheduler	Description	Source File
	traversing the request queue searching for optimal insertion points. Instead, it simply adds new requests to the tail of the request queue. This scheduler is thus ideal for semiconductor storage media that have no moving parts and, hence, no seek latencies. An example is a *Disk-On-Module* (DOM), which internally uses flash memory.	(in the 2.6 kernel tree)

At a conceptual level, I/O scheduling resembles process scheduling. Whereas I/O scheduling provides an illusion to processes that they own the disk, process scheduling gives processes the illusion that they own the CPU. Both I/O and process schedulers on Linux have undergone extensive changes in recent times. Process scheduling is discussed in Chapter 19.

Block Driver Data Structures and Methods

Let's now shift focus to the main topic of this chapter, block device drivers. In this section, we take a look at the important data structures and driver methods that you are likely to encounter while implementing a block device driver. We use these structures and methods in the next section when we implement a block driver for a fictitious storage controller.

The following are the main block driver data structures:

1. The kernel represents a disk using the `gendisk` (short for generic disk) structure defined in *include/linux/genhd.h*:

```
struct gendisk {
  int major;                      /* Device major number */
  int first_minor;                /* Starting minor number */
  int minors;                     /* Maximum number of minors.
                                     You have one minor number
                                     per disk partition */
  char disk_name[32];             /* Disk name */
  /* ... */
  struct block_device_operations *fops;
                                  /* Block device operations.
                                     Described soon. */
  struct request_queue *queue;    /* The request queue associated
                                     with this disk. Discussed
                                     next. */
  /* ... */
};
```

2. The I/O request queue associated with each block driver is described using the `request_queue` structure defined in *include/linux/blkdev.h*. This is a big structure, but its only constituent field that you might use is the `request` structure, which is described next.

3. Each request in a `request_queue` is represented using a `request` structure defined in *include/linux/blkdev.h*:

```
struct request {
  /* ... */
  struct request_queue *q;  /* The container request queue */
  /* ... */
  sector_t sector;          /* Sector from which data access
                               is requested */
  /* ... */
  unsigned long nr_sectors; /* Number of sectors left to
                               submit */
  /* ... */
  struct bio *bio;          /* The associated bio. Discussed
                               soon. */
  /* ... */
  char *buffer;             /* The buffer for data transfer */
```

```
  /* ... */
  struct request *next_rq;  /* Next request in the queue */
};
```

4. `block_device_operations` is the block driver counterpart of the `file_operations` structure used by character drivers. It contains the following entry points associated with a block driver:

   - Standard methods such as `open()`, `release()`, and `ioctl()`

   - Specialized methods such as `media_changed()` and `revalidate_disk()` that support removable block devices

     `block_device_operations` is defined as follows in *include/linux/fs.h*:

```
struct block_device_operations {
  int (*open) (struct inode *, struct file *);   /* Open */
  int (*release) (struct inode *, struct file *);/* Close */
  int (*ioctl) (struct inode *, struct file *,
                unsigned, unsigned long);    /* I/O Control */
  /* ... */
  int (*media_changed) (struct gendisk *);   /* Check if media is
                                                available or
                                                ejected */
  int (*revalidate_disk) (struct gendisk *); /* Gear up for newly
                                                inserted media */
  /* ... */
};
```

5. When we looked at the `request` structure, we saw that it was associated with a `bio`. A `bio` structure is a low-level description of block I/O operations at page-level granularity. It's defined in *include/linux/bio.h* as follows:

```
struct bio {
  sector_t        bi_sector;  /* Sector from which data
                                 access is requested */
  struct bio     *bi_next;    /* List of bio nodes */
  /* .. */
  unsigned long   bi_rw;      /* Bottom bits of bi_rw contain
                                 the data-transfer direction */
  /* ... */
  struct bio_vec *bi_io_vec;  /* Pointer to an array of
                                 bio_vec structures */
  unsigned short  bi_vcnt;    /* Size of the bio_vec array */
  unsigned short  bi_idx;     /* Index of the current bio_vec
                                 in the array */
  /* ... */
};
```

   Block data is internally represented as an I/O vector using an array of `bio_vec` structures. Each element of the `bio_vec` array is made up of a `(page, page_offset, length)` tuple that describes a segment of the I/O block. Maintaining I/O requests as a vector of pages brings several advantages, including a leaner implementation and efficient scatter/gather.

Before ending this section, let's briefly look at block driver entry points. Block drivers are broadly built using three types of methods:

- The usual initialization and exit methods.
- Methods that are part of the `block_device_operations` described previously.
- A request method. Block drivers, unlike char devices, do not support `read()/write()` methods for data transfer. Instead, they perform disk access using a special routine called the *request method*.

The block core layer offers a set of library routines that driver methods can leverage. The sample driver in the next section calls on the services of several of these library routines.

Device Example: Simple Storage Controller

Consider the embedded device shown in Figure 14.3. The SoC contains a built-in storage controller that communicates with a block device. The architecture is similar to SD/MMC media, but our sample storage controller is described by the elementary register set listed in Table 14.3. The `SECTOR_NUMBER_REGISTER` specifies the sector from which data access is requested.[4] The `SECTOR_COUNT_REGISTER` contains the number of sectors to be transferred. Data is moved via the `DATA_REGISTER`. The `COMMAND_REGISTER` programs the action that the storage controller has to take (for example, whether to read from the media or write to it). The `STATUS_REGISTER` contains bits that signal whether the controller is busy performing an operation.

[4] The storage media in our sample device has a flat sector-space geometry. IDE controllers, on the other hand, support a *cylinder head sector* (CHS) geometry specified by a device head register, a low cylinder register, and a high cylinder register, in addition to the sector number register.

Figure 14.3. Storage on an embedded device.

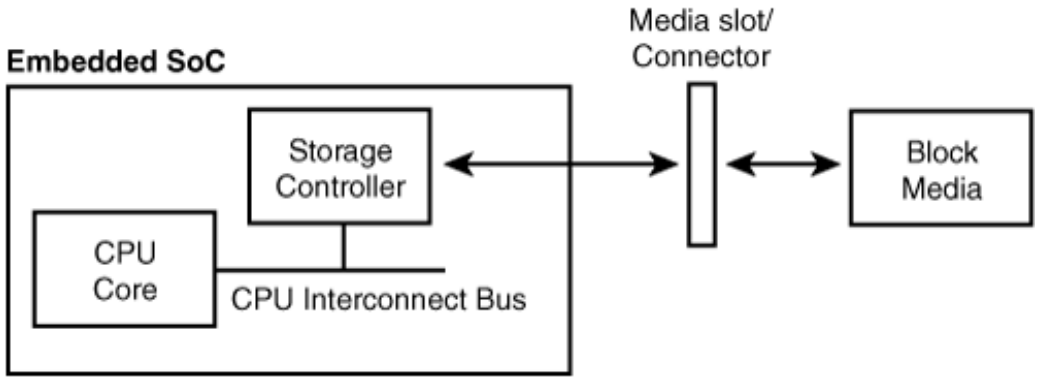


Table 14.3. Register Layout of the Storage Controller

Register Name	Description of Contents
`SECTOR_NUMBER_REGISTER`	The sector on which the next disk operation is to be performed.
`SECTOR_COUNT_REGISTER`	Number of sectors to be read or written.
`COMMAND_REGISTER`	The action to be taken (for example, read or write).
`STATUS_REGISTER`	Results of operations, interrupt status, and error flags.
`DATA_REGISTER`	In the read path, the storage controller fetches data from the disk to internal buffers. The driver accesses the internal buffer via this register. In the write path, data written by the driver to this register is transferred to the internal buffer, from where the controller copies it to disk.

Let's call the storage controller *myblkdev*. This simple device is neither interrupt driven nor supports DMA. We'll also assume that the media is not removable. Our task is to write a block driver for myblkdev. Our driver is minimal, albeit complete. It does not handle power management and is not particularly performance-sensitive.

Initialization

Listing 14.1 contains the driver initialization method, `myblkdev_init ()`, which performs the following steps:

1. Registers the block device using `register_blkdev()`. This block library routine assigns an unused major number to myblkdev and adds an entry for the device in */proc/devices*.

2. Associates a request method with the block device. It does this by supplying the address of `myblkdev_request()` to `blk_init_queue()`. The call to `blk_init_queue()` returns the `request_queue` for myblkdev. Refer back to Figure 14.2 to see how the `request_queue` sits relative to the driver. The second argument to `blk_init_queue()`, `myblkdev_lock`, is a spinlock to protect the `request_queue` from concurrent access.

3. Hardware performs disk transactions in units of *sectors*, whereas software subsystems, such as filesystems, deal with data in terms of *blocks*. The common sector size is 512 bytes; the usual block size is 4096 bytes. You need to inform the block layer about the sector size supported by your storage hardware and the maximum number of sectors that your driver can receive per request. `myblkdev_init()` accomplishes these by invoking `blk_queue_hardsect_size()` and `blk_queue_max_sectors()`, respectively.

4. Allocates a `gendisk` corresponding to myblkdev using `alloc_disk()` and populates it. One important `gendisk` field that `myblkdev_init()` supplies is the address of the driver's `block_device_operations`. Another parameter that `myblkdev_init()` fills in is the storage capacity of myblkdev in units of sectors. This is accomplished by calling `set_capacity()`. Each `gendisk` also contains a flag that signals the properties of the underlying storage hardware. If the drive is removable, for example, the `gendisk`'s `flag` field should be marked `GENHD_FL_REMOVABLE`.

5. Associates the `gendisk` prepared in Step 4 with the `request_queue` obtained in Step 2. Also, connects the `gendisk` with the device's major/minor numbers and a name.

6. Adds the disk to the block I/O layer by invoking `add_disk()`. When this is done, the driver has to be ready to receive requests. So, this is usually the last step of the initialization sequence.

The block device is now available to the system as */dev/myblkdev*. If the device supports multiple disk partitions, they appear as */dev/myblkdevX*, where *X* is the partition number.

Listing 14.1. Initializing the Driver

Code View:

```
#include <linux/blkdev.h>
#include <linux/genhd.h>

static struct gendisk *myblkdisk;     /* Representation of a disk */
static struct request_queue *myblkdev_queue;
                                      /* Associated request queue */
int myblkdev_major = 0;               /* Ask the block subsystem
                                         to choose a major number */
static DEFINE_SPINLOCK(myblkdev_lock);/* Spinlock that protects
                                         myblkdev_queue from
                                         concurrent access */
```

```
int myblkdisk_size     = 256*1024;    /* Disk size in kilobytes. For
                                         a PC hard disk, one way to
                                         glean this is via the BIOS */
int myblkdev_sect_size = 512;         /* Hardware sector size */
/* Initialization */
static int __init
myblkdev_init(void)
{
  /* Register this block driver with the kernel */
  if ((myblkdev_major = register_blkdev(myblkdev_major,
                                        "myblkdev")) <= 0) {
    return -EIO;
  }

  /* Allocate a request_queue associated with this device */
  myblkdev_queue = blk_init_queue(myblkdev_request, &myblkdev_lock);
  if (!myblkdev_queue) return -EIO;

  /* Set the hardware sector size and the max number of sectors */
  blk_queue_hardsect_size(myblkdev_queue, myblkdev_sect_size);
  blk_queue_max_sectors(myblkdev_queue, 512);

  /* Allocate an associated gendisk */
  myblkdisk = alloc_disk(1);
  if (!myblkdisk) return -EIO;

  /* Fill in parameters associated with the gendisk */
  myblkdisk->fops   = &myblkdev_fops;

  /* Set the capacity of the storage media in terms of number of
     sectors */
  set_capacity(myblkdisk, myblkdisk_size*2);

  myblkdisk->queue  = myblkdev_queue;
  myblkdisk->major = myblkdev_major;
  myblkdisk->first_minor = 0;
  sprintf(myblkdisk->disk_name, "myblkdev");

  /* Add the gendisk to the block I/O subsystem */
  add_disk(myblkdisk);

  return 0;
}

/* Exit */
static void __exit
myblkdev_exit(void)
{
  /* Invalidate partitioning information and perform cleanup */
  del_gendisk(myblkdisk);

  /* Drop references to the gendisk so that it can be freed */
  put_disk(myblkdisk);

  /* Dissociate the driver from the request_queue. Internally calls
     elevator_exit() */
  blk_cleanup_queue(myblkdev_queue);
```

```
  /* Unregister the block device */
  unregister_blkdev(myblkdev_major, "myblkdev");
}

module_init(myblkdev_init);
module_exit(myblkdev_exit);
MODULE_LICENSE("GPL");
```

Block Device Operations

Let's next take a look at the main methods contained in a block driver's `block_device_operations`.

A block driver's `open()` method is called during operations such as mounting a filesystem residing on the media or performing a filesystem check (fsck). Many of the tasks accomplished during `open()` are hardware-dependent. The CD-ROM driver, for example, locks the drive door. The SCSI driver checks whether the device has set a write-protect tab, and, if so, fails if a write-enabled open is requested. If the device is removable, `open()` invokes the service routine `check_disk_change()` to check whether the media has changed.

If your driver needs to support specific commands, implement support for it using the `ioctl()` method. A floppy driver, for example, supports a command to eject the media.

The `media_changed()` method checks whether the storage media has changed, so this is not relevant for fixed devices such as myblkdev. The SCSI disk driver's `media_changed()` method, for example, detects whether an inserted USB pen drive has changed.

The sole block device operation supported by myblkdev is the `ioctl()` method, `myblkdev_ioctl()`. The block layer itself handles generic ioctls and invokes the driver's `ioctl()` method only to handle device-specific commands. In Listing 14.2, `myblkdev_ioctl()` implements the `GET_DEVICE_ID` command that elicits a device ID from the controller. The command is issued via the `COMMAND_REGISTER`, and the ID data is obtained from the `DATA_REGISTER`.

Listing 14.2. Block Device Operations

Code View:

```
#define GET_DEVICE_ID 0xAA00   /* Ioctl command definition */

/* The ioctl operation */
static int
myblkdev_ioctl (struct inode *inode, struct file *file,
                unsigned int cmd, unsigned long arg)
{
  unsigned char status;

  switch (cmd) {
  case GET_DEVICE_ID:
    outb(GET_IDENTITY_CMD, COMMAND_REGISTER);
    /* Wait as long as the controller is busy */
    while ((status = inb(STATUS_REGISTER)) & BUSY_STATUS);

    /* Obtain ID and return it to user space */
    return put_user(inb(DATA_REGISTER), (long __user *)arg);
  default:
    return -EINVAL;
  }
}

/* Block device operations */
static struct block_device_operations myblkdev_fops = {
  .owner = THIS_MODULE,        /* Owner of this structure */
  .ioctl = myblkdev_ioctl,
  /* The following operations are not implemented for our example
     storage controller: open(), release(), unlocked_ioctl(),
     compat_ioctl(), direct_access(), getgeo(), revalidate_disk(), and
     media_changed() */
};
```

Disk Access

As mentioned previously, block drivers perform disk access operations using a `request()` method. The block I/O subsystem invokes a driver's `request()` method whenever it desires to process requests waiting in the driver's `request_queue`. The `request()`method does not run in the context of the user process requesting the data transfer, however. The address of the associated `request_queue` is passed as an argument to the `request()` method.

As you saw earlier, the kernel holds a request lock before calling the `request()` method. This is to protect the associated request queue from concurrent access. Because of this, if your `request()` method has to call any functions that may go to sleep, it has to drop the lock before doing so and reacquire it before returning.

Listing 14.3 contains our driver's request method, `myblkdev_request()`. This function uses the services of `elv_next_request()` to obtain the next request from the `request_queue`. If the queue contains no more pending requests, `elv_next_request()` returns `NULL`. `elv_next_request()` is named so because, as you learned previously, I/O scheduling algorithms are variations of the basic modus operandi adopted by elevators to service requests. After handling a request, the driver asks the block layer to end I/O on that request by calling `end_request()`. You can specify success or an error code using the second argument to `end_request()`.

Requests collected from the `request_queue` contain the starting sector from which data access is requested (`req->sector` in Listing 14.3, the number of sectors on which I/O needs to be performed (`req->nr_sectors`), the buffer that contains the data to be transferred (`req->buffer`), and the direction of data movement

(rq_data_dir(req)). As shown in Listing 14.3, myblkdev_request() performs the required register programming with the help of these parameters.

Listing 14.3. The Request Function

Code View:

```
#define READ_SECTOR_CMD           1
#define WRITE_SECTOR_CMD          2
#define GET_IDENTITY_CMD          3

#define BUSY_STATUS               0x10

#define SECTOR_NUMBER_REGISTER    0x20000000
#define SECTOR_COUNT_REGISTER     0x20000001
#define COMMAND_REGISTER          0x20000002
#define STATUS_REGISTER           0x20000003
#define DATA_REGISTER             0x20000004

/* Request method */
static void
myblkdev_request(struct request_queue *rq)
{
  struct request *req;
  unsigned char status;
  int i, good = 0;

  /* Loop through the requests waiting in line */
  while ((req = elv_next_request(rq)) != NULL) {
    /* Program the start sector and the number of sectors */
    outb(req->sector, SECTOR_NUMBER_REGISTER);
    outb(req->nr_sectors, SECTOR_COUNT_REGISTER);

    /* We are interested only in filesystem requests. A SCSI command
       is another possible type of request. For the full list, look
       at the enumeration of rq_cmd_type_bits in
       include/linux/blkdev.h */
     if (blk_fs_request(req)) {
       switch(rq_data_dir(req)) {
       case READ:
           /* Issue Read Sector Command */
              outb(READ_SECTOR_CMD, COMMAND_REGISTER);
           /* Traverse all requested sectors, byte by byte */
              for (i = 0; i < 512*req->nr_sectors; i++) {
           /* Wait until the disk is ready. Busy duration should be
              in the order of microseconds. Sitting in a tight loop
              for simplicity; more intelligence required in the real
              world */
             while ((status = inb(STATUS_REGISTER)) & BUSY_STATUS);

             /* Read data from disk to the buffer associated with the
                request */
             req->buffer[i] = inb(DATA_REGISTER);
           }
         good = 1;
         break;
       case WRITE:
        /* Issue Write Sector Command */
        outb(WRITE_SECTOR_CMD, COMMAND_REGISTER);
```

```
        /* Traverse all requested sectors, byte by byte */
        for (i = 0; i < 512*req->nr_sectors; i++) {
          /* Wait until the disk is ready. Busy duration should be
             in the order of microseconds. Sitting in a tight loop
             for simplicity; more intelligence required in the real
             world */
           while ((status = inb(STATUS_REGISTER)) & BUSY_STATUS);

          /* Write data to disk from the buffer associated with the
             request */
          outb(req->buffer[i], DATA_REGISTER);
        }
        good = 1;
        break;
      }
    }
    end_request(req, good);
  }
}
```

Advanced Topics

Unlike our sample storage driver that transfers data byte by byte, performance-sensitive block drivers rely on DMA for data transfer. Consider, for example, the `request()` method of the disk array driver for Compaq SMART2 controllers *drivers/block/-cpqarray.c* reproduced here from the 2.6.23.1 kernel sources:

Code View:
```
static do_ida_request(struct request_queue *q)
{
  struct request *creq;
  struct scatterlist tmp_sg[SG_MAX];
  cmdlist_t *c;
  ctrl_info_t *h = q->queuedata;
  int seg;

  /* ... */
  creq = elv_next_request(q);
  /* ... */
  c->rq = creq;
  seg = blk_rq_map_sg(q, creq, tmp_sg);
  /* ... */
  for (i=0; i<seg; i++)
  {
    c->req.sg[i].size = tmp_sg[i].length;
    c->req.sg[i].addr = (__u32) pci_map_page(h->pci_dev,
                                             tmp_sg[i].page,
                                             tmp_sg[i].offset,
                                             tmp_sg[i].length, dir);
  }
  /* ... */
}
```

DMA operations work at `bio` level. As you saw earlier, I/O requests are made up of `bio`s, each of which contains an array of `bio_vec`s, which in turn hold information about the constituent memory pages. Assuming that `bio` points to the `bio` structure associated with an I/O request, `bio->bi_sector` contains the starting sector from which data access is requested, `bio_cur_sectors(bio)` returns the number of sectors on which I/O is to be performed, and `bio_data_dir(bio)` provides the direction of data transfer. The addresses of the physical pages associated with the data buffer are described by the array of `bio_vec`s pointed to by `bio->bi_io_vec`. To iterate over each `bio` in a request, you can use the `rq_for_each_bio()`macro. To further loop through each page segment in a `bio`, use `bio_for_each_segment()`.

In the preceding code snippet, `blk_rq_map_sg()` internally invokes `rq_for_each_bio()` and `bio_for_each_segment()`to loop through all pages constituting the request and builds a scatter/gather list, `tmp_sg`. Streaming DMA mappings for each page in the created scatter/gather list is performed by `pci_map_page()`.

Unlike our sample driver that busy-waits for requested operations to finish, the *cpqarray* driver implements an interrupt handler, `do_ida_intr()`, to receive alerts from the hardware upon completion of commands.

Some drivers, such as the ramdisk driver (*drivers/block/rd.c*) and the loopback driver (*drivers/block/loop.c*),

work over virtual block devices that do not benefit from the optimizing sort and merge operations on the request queue. Such drivers entirely bypass the request queue and directly obtain `bio`s from the block layer using a `make_request()` function. So, instead of registering a request queue handler using `blk_init_queue()`, *drivers/block/rd.c* supplies a `make_request()` routine using `blk_queue_make_request()` as follows:

```
static int __init rd_init(void)
{
  /* ... */
  blk_queue_make_request(rd_queue[i], &rd_make_request);
  /* ... */
}

static int rd_make_request(struct request_queue *q, struct bio *bio)
{
  /* ... */
}
```

Debugging

The *hdparm* utility elicits various PATA/SATA disk parameters from the underlying kernel driver. To benchmark disk read speeds on a SATA drive, for example, do this:

```
bash> hdparm -T -t /dev/sda
/dev/sda:
 Timing cached reads: 2564 MB in 2.00 seconds = 1283.57 MB/sec
 Timing buffered disk reads: 132 MB in  3.03 seconds = 43.61 MB/sec
```

For the full capabilities of hdparm, read the man pages.

Self-Monitoring, Analysis, *and Reporting Technology* (SMART) is a system built in to many modern ATA and SCSI disks to monitor failures and perform self-tests. A user-space daemon named *smartd* collects the information gathered by SMART-capable disks with the help of the underlying device driver. Look at the man pages of smartd, smartctl, and smartd.conf to learn how to obtain health status from SMART-enabled disks.

If your distribution doesn't prepackage hdparm and SMART tools, you may download them from http://sourceforge.net/projects/hdparm/ and http://sourceforge.net/projects/smartmontools/, respectively.

Files under */proc/ide/* contain information about IDE disk drives on your system. To obtain the geometry of the first IDE disk, for example, look at the contents of */proc/ide/ide0/hda/geometry*. Information pertaining to SCSI devices is available under */proc/scsi/*. You can gather disk partition information from */proc/partitions*.

The sysfs directory of interest for IDE devices is */sys/bus/ide/* and for SCSI is */sys/bus/scsi/*. In addition, each block device active on the system owns a subdirectory under */sys/block/*, which contains associated request queue parameters, constituent partition details, and state information.

Some kernel configuration options are available that trigger the emission of debug output from the block subsystem. `CONFIG_BLK_DEV_IO_TRACE` provides the ability to trace the block layer. `CONFIG_SCSI_CONSTANTS` and `CONFIG_SCSI_LOGGING` turn on SCSI error reporting and logging, respectively.

The linux-ide mailing list is the forum to discuss questions related to the Linux-IDE subsystem. Subscribe to the linux-scsi mailing list and browse through its archives for discussions pertaining to the Linux-SCSI subsystem.

Looking at the Sources

Table 14.1 contains the location of kernel driver sources for various storage technologies. Take a look at *Documentation/ide.txt*, *Documentation/scsi/**, and *Documentation/cdrom/* for information about associated storage drivers.

The top-level *block/* directory contains I/O scheduling algorithms and the block core layer. Table 14.2 lists the source files in this directory that implement various I/O schedulers. Look at *Documentation/block/* for related documentation.

Table 14.4 contains the main data structures used in this chapter and their location in the source tree. Table 14.5 lists the main kernel programming interfaces that you used in this chapter, along with the location of their definitions.

Table 14.4. Summary of Data Structures

Data Structure	Location	Description
`gendisk`	*include/linux/genhd.h*	Representation of a disk.
`request_queue`	*include/linux/blkdev.h*	The I/O request queue associated with a `gendisk`.
`request`	*include/linux/blkdev.h*	Each request in a `request_queue` is described using this structure.
`block_device_operations`	*include/linux/fs.h*	Block device driver methods.
`bio`	*include/linux/bio.h*	Low-level description of block I/O operations.

Table 14.5. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`register_blkdev()`	*block/genhd.c*	Registers a block driver with the kernel
`unregister_blkdev()`	*block/genhd.c*	Unregisters a block driver from the kernel
`alloc_disk()`	*block/genhd.c*	Allocates a `gendisk`
`add_disk()`	*block/genhd.c*	Adds a populated `gendisk` to the kernel block layer
`del_gendisk()`	*fs/partitions/check.c*	Frees a `gendisk`
`blk_init_queue()`	*block/ll_rw_blk.c*	Allocates a `request_queue` and registers a `request()` function to process the requests in the queue
`blk_cleanup_queue()`	*block/ll_rw_blk.c*	Reverse of `blk_init_queue()`

Kernel Interface	Location	Description
`blk_queue_make_request()`	*block/ll_rw_blk.c*	Registers a `make_request()` function, which bypasses the request queue and directly obtains requests from the block layer
`rq_for_each_bio()`	*include/linux/blkdev.h*	Iterates over each `bio` in a request
`bio_for_each_segment()`	*include/linux/bio.h*	Loops through each page segment in a `bio`
`blk_rq_map_sg()`	*block/ll_rw_blk.c*	Iterates through the `bio` segments constituting a request and builds a scatter/gather list
`blk_queue_max_sectors()`	*block/ll_rw_blk.c*	Sets the maximum sectors for a request in the associated request queue
`blk_queue_hardsect_size()`	*block/ll_rw_blk.c*	Sector size supported by the storage hardware.
`set_capacity()`	*include/linux/genhd.h*	Sets the capacity of the storage media in terms of number of sectors
`blk_fs_request()`	*include/linux/blkdev.h*	Checks whether a request obtained from the request queue is a filesystem request
`elv_next_request()`	*block/elevator.c*	Obtains the next entry from the request queue
`end_request()`	*block/ll_rw_blk.c*	Ends I/O on a request

Chapter 15. Network Interface Cards

In This Chapter

- Driver Data Structures 440
- Talking with Protocol Layers 448
- Buffer Management and Concurrency Control 450
- Device Example: Ethernet NIC 451
- ISA Network Drivers 457
- Asynchronous Transfer Mode 458
- Network Throughput 459
- Looking at the Sources 461

Connectivity imparts intelligence. You rarely come across a computer system today that does not support some form of networking. In this chapter, let's focus on device drivers for *network interface cards* (NICs) that carry *Internet Protocol* (IP) traffic on a *local area network* (LAN). Most of the chapter is bus agnostic, but wherever bus specifics are necessary, it assumes PCI. To give you a flavor of other network technologies, we also touch on *Asynchronous Transfer Mode* (ATM). We end the chapter by pondering on performance and throughput.

NIC drivers are different from other driver classes in that they do not rely on */dev* or */sys* to communicate with user space. Rather, applications interact with a NIC driver via a network interface (for example, eth0 for the first Ethernet interface) that abstracts an underlying protocol stack.

Driver Data Structures

When you write a device driver for a NIC, you have to operate on three classes of data structures:

1. Structures that form the building blocks of the network protocol stack. The socket buffer or `struct sk_buff` defined in *include/linux/sk_buff.h* is the key structure used by the kernel's TCP/IP stack.

2. Structures that define the interface between the NIC driver and the protocol stack. `struct net_device` defined in *include/linux/netdevice.h* is the core structure that constitutes this interface.

3. Structures related to the I/O bus. PCI and its derivatives are common buses used by today's NICs.

We take a detailed look at socket buffers and the `net_device` interface in the next two sections. We covered PCI data structures in Chapter 10, "Peripheral Component Interconnect," so we won't revisit them here.

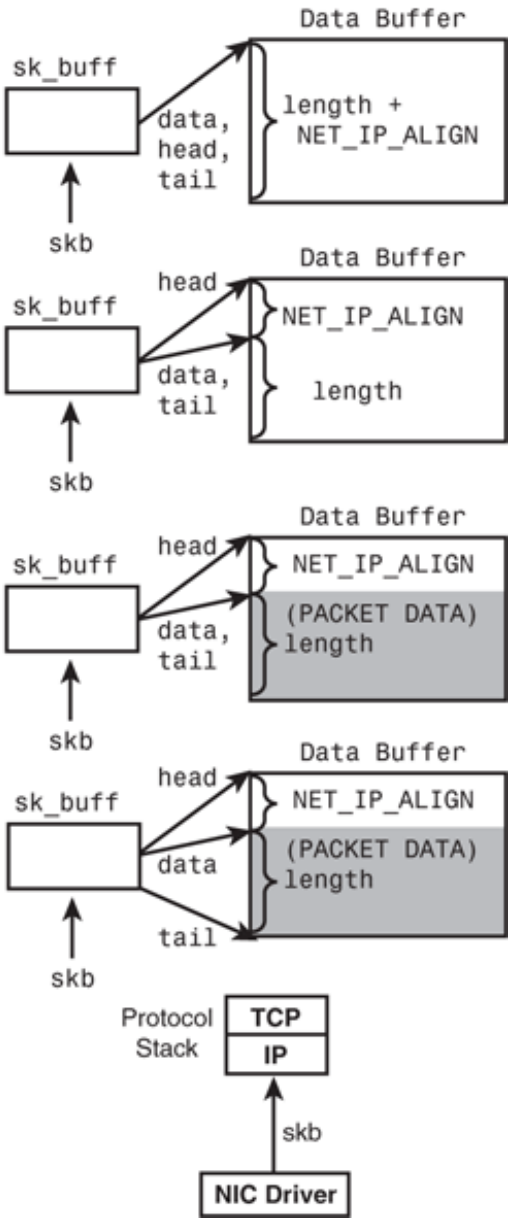
Socket Buffers

`sk_buff`s provide efficient buffer handling and flow-control mechanisms to Linux networking layers. Like DMA descriptors that contain metadata on DMA buffers, `sk_buff`s hold control information describing attached memory buffers that carry network packets (see Figure 15.1). `sk_buff`s are enormous structures having dozens of elements, but in this chapter we confine ourselves to those that interest the network device driver writer. An `sk_buff` links itself to its associated packet buffer using five main fields:

- `head`, which points to the start of the packet
- `data`, which points to the start of packet payload
- `tail`, which points to the end of packet payload
- `end`, which points to the end of the packet
- `len`, the amount of data that the packet contains

Figure 15.1. **`sk_buff`** operations.

```
struct sk_buff  *skb;
/* ... */

skb = dev_alloc_skb(length +
                    NET_IP_ALIGN);

skb_reserve(skb, NET_IP_ALIGN);

mempcy(skb->data, dma_buffer,
       length)

skb_put(skb, length);

netif_rx(skb);
```



Assume `skb` points to an `sk_buff`, `skb->head`, `skb->data`, `skb->tail`, and `skb->end` slide over the associated packet buffer as the packet traverses the protocol stack in either direction. `skb->data`, for example, points to the header of the protocol that is currently processing the packet. When a packet reaches the IP layer via the receive path, `skb->data` points to the IP header; when the packet passes on to TCP, however, `skb->data` moves to the start of the TCP header. And as the packet drives through various protocols adding or discarding header data, `skb->len` gets updated, too. `sk_buff`s also contain pointers other than the four major ones previously mentioned. `skb->nh`, for example, remembers the position of the network protocol header irrespective of the current position of `skb->data`.

To illustrate how a NIC driver works with `sk_buff`s, Figure 15.1 shows data transitions on the receive data path. For convenience of illustration, the figure simplistically assumes that the operations shown are executed in sequence. However, for operational efficiency in the real world, the first two steps (`dev_alloc_skb()` and

skb_reserve()) are performed while initially preallocating a ring of receive buffers; the third step is accomplished by the NIC hardware as it directly DMA's the received packet into a preallocated sk_buff; and the final two steps (skb_put() and netif_rx()) are executed from the receive interrupt handler.

To create an sk_buff to hold a received packet, Figure 15.1 uses dev_alloc_skb(). This is an interrupt-safe routine that allocates memory for an sk_buff and associates it with a packet payload buffer. dev_kfree_skb() accomplishes the reverse of dev_alloc_skb(). Figure 15.1 next calls skb_reserve() to add a 2-byte padding between the start of the packet buffer and the beginning of the payload. This starts the IP header at a performance-friendly 16-byte boundary because the preceding Ethernet headers are 14 bytes long. The rest of the code statements in Figure 15.1 fill the payload buffer with the received packet and move skb->data, skb->tail, and skb->len to reflect this operation.

There are more sk_buff access routines relevant to some NIC drivers. skb_clone(), for example, creates a copy of a supplied skb_buff without copying the contents of the associated packet buffer. Look inside *net/core/skbuff.c* for the full list of sk_buff library functions.

The Net Device Interface

NIC drivers use a standard interface to interact with the TCP/IP stack. The net_device structure, which is even more gigantic than the sk_buff structure, defines this communication interface. To prepare ourselves for exploring the innards of the net_device structure, let's first follow the steps traced by a NIC driver during initialization. Refer to init_mycard() in Listing 15.1 as we move along:

- The driver allocates a net_device structure using alloc_netdev(). More commonly, it uses a suitable wrapper around alloc_netdev(). An Ethernet NIC driver, for example, calls alloc_etherdev(). A WiFi driver (discussed in the next chapter) invokes alloc_ieee80211(), and an IrDa driver calls upon alloc_irdadev(). All these functions take the size of a private data area as argument and create this area in addition to the net_device itself:

```
struct net_device  *netdev;
struct priv_struct *mycard_priv;
netdev      = alloc_etherdev(sizeof(struct
                             priv_struct));
mycard_priv = netdev->priv; /* Private area created
                               by alloc_etherdev() */
```

- Next, the driver populates various fields in the net_device that it allocated and registers the populated net_device with the network layer using register_netdev(netdev).

- The driver reads the NIC's *Media Access Control* (MAC) address from an accompanying EEPROM and configures *Wake-On-LAN* (WOL) if required. Ethernet controllers usually have a companion nonvolatile EEPROM to hold information such as their MAC address and WOL pattern, as shown in Figure 15.2. The former is a unique 48-bit address that is globally assigned. The latter is a magic sequence; if found in received data, it rouses the NIC if it's in suspend mode.

- If the NIC needs on-card firmware to operate, the driver downloads it using request_firmware(), as discussed in the section "Microcode Download" in Chapter 4, "Laying the Groundwork."

Let's now look at the methods that define the net_device interface. We categorize them under six heads for simplicity. Wherever relevant, this section points you to the example NIC driver developed in Listing 15.1 of the section "Device Example: Ethernet NIC."

Activation

The net_device interface requires conventional methods such as open(), close(), and ioctl(). The kernel opens an interface when you activate it using a tool such as *ifconfig:*

```
bash> ifconfig eth0 up
```

open() sets up receive and transmit DMA descriptors and other driver data structures. It also registers the NIC's interrupt handler by calling request_irq(). The net_device structure is passed as the devid argument to request_irq() so that the interrupt handler gets direct access to the associated net_device. (See mycard_open() and mycard_interrupt() in Listing 15.1 to find out how this is done.)

The kernel calls close() when you pull down an active network interface. This accomplishes the reverse of open().

Data Transfer

Data transfer methods form the crux of the net_device interface. In the transmit path, the driver supplies a method called hard_start_xmit, which the protocol layer invokes to pass packets down for onward transmission:

Code View:
```
netdev->hard_start_xmit = &mycard_xmit_frame; /* Transmit Method. See Listing 15.1 */
```

Until recently, network drivers didn't provide a net_device method for collecting received data. Instead, they asynchronously interrupted the protocol layer with packet payload. This old interface has, however, given way to a *New API* (NAPI) that is a mixture of an interrupt-driven driver push and a poll-driver protocol pull. A NAPI-aware driver thus needs to supply a poll() method and an associated weight that controls polling fairness:

```
netdev->poll   = &mycard_poll; /* Poll Method. See Listing 15.1 */
netdev->weight = 64;
```

We elaborate on data-transfer methods in the section "Talking with Protocol Layers."

Watchdog

The net_device interface provides a hook to return an unresponsive NIC to operational state. If the protocol layer senses no transmissions for a predetermined amount of time, it assumes that the NIC has hung and invokes a driver-supplied recovery method to reset the card. The driver sets the watchdog timeout through netdev->watchdog_timeo and registers the address of the recovery function via netdev->tx_timeout:

```
netdev->tx_timeout = &mycard_timeout; /* Method to reset the NIC */
netdev->watchdog_timeo = 8*HZ;        /* Reset if no response
                                         detected for 8 seconds */
```

Because the recovery method executes in timer-interrupt context, it usually schedules a task outside of that context to reset the NIC.

Statistics

To enable user land to collect network statistics, the NIC driver populates a net_device_stats structure and

provides a `get_stats()` method to retrieve it. Essentially the driver does the following:

1. Updates different types of statistics from relevant entry points:

```
#include <linux/netdevice.h>
struct net_device_stats mycard_stats;

static irqreturn_t
mycard_interrupt(int irq, void *dev_id)
{
  /* ... */
  if (packet_received_without_errors) {
    mycard_stats.rx_packets++;   /* One more received
                                    packet */
  }
  /* ... */
}
```

2. Implements the `get_stats()` method to retrieve the statistics:

```
static struct net_device_stats
*mycard_get_stats(struct net_device *netdev)
{
   /* House keeping */
   /* ... */
   return(&mycard_stats);
}
```

3. Supplies the retrieve method to higher layers:

```
netdev->get_stats = &mycard_get_stats;
/* ... */
register_netdev(netdev);
```

To collect statistics from your NIC, trigger invocation of `mycard_get_stats()` by executing an appropriate user mode command. For example, to find the number of packets received through the eth0 interface, do this:

```
bash> cat /sys/class/net/eth0/statistics/rx_packets
124664
```

WiFi drivers need to track several parameters not relevant to conventional NICs, so they implement a statistic collection method called `get_wireless_stats()` in addition to `get_stats()`. The mechanism for registering `get_wireless_stats()` for the benefit of WiFi-aware user space utilities is discussed in the section "WiFi" in the next chapter.

Configuration

NIC drivers need to support user space tools that are responsible for setting and getting device parameters. *Ethtool* configures parameters for Ethernet NICs. To support ethtool, the underlying NIC driver does the following:

1. Populates an `ethtool_ops` structure, defined in *include/linux/ethtool.h* with prescribed entry points:

```
#include <linux/ethtool.h>

/* Ethtool_ops methods */
struct ethtool_ops mycard_ethtool_ops = {
  /* ... */
  .get_eeprom = mycard_get_eeprom, /* Dump EEPROM
                                      contents */
  /* ... */
};
```

2. Implements the methods that are part of `ethtool_ops`:

```
static int
mycard_get_eeprom(struct net_device *netdev,
                  struct ethtool_eeprom *eeprom,
                  uint8_t *bytes)
{
   /* Access the accompanying EEPROM and pull out data */
   /* ... */
}
```

3. Exports the address of its `ethtool_ops`:

```
netdev->ethtool_ops = &mycard_ethtool_ops;
/* ... */
register_netdev(netdev);
```

After these are done, ethtool can operate over your Ethernet NIC. To dump EEPROM contents using ethtool, do this:

```
bash> ethtool -e eth0
Offset          Values
------          ------
0x0000          00 0d 60 79 32 0a 00 0b ff ff 10 20 ff ff ff ff
...
```

Ethtool comes packaged with some distributions; but if you don't have it, download it from http://sourceforge.net/projects/gkernel/. Refer to the man page for its full capabilities.

There are more configuration-related methods that a NIC driver provides to higher layers. An example is the method to change the MTU size of the network interface. To support this, supply the relevant method to `net_device`:

```
netdev->change_mtu = &mycard_change_mtu;
/* ... */
register_netdev(netdev);
```

The kernel invokes `mycard_change_mtu()` when you execute a suitable user command to alter the MTU of your card:

```
bash> echo 1500 > /sys/class/net/eth0/mtu
```

Bus Specific

Next come bus-specific details such as the start address and size of the NIC's on-card memory. For a PCI NIC driver, this configuration will look like this:

```
netdev->mem_start = pci_resource_start(pdev, 0);
netdev->mem_end   = netdev->mem_start + pci_resource_len(pdev, 0);
```

We discussed PCI resource functions in Chapter 10.

Chapter 15. Network Interface Cards

In This Chapter

- Driver Data Structures 440
- Talking with Protocol Layers 448
- Buffer Management and Concurrency Control 450
- Device Example: Ethernet NIC 451
- ISA Network Drivers 457
- Asynchronous Transfer Mode 458
- Network Throughput 459
- Looking at the Sources 461

Connectivity imparts intelligence. You rarely come across a computer system today that does not support some form of networking. In this chapter, let's focus on device drivers for *network interface cards* (NICs) that carry *Internet Protocol* (IP) traffic on a *local area network* (LAN). Most of the chapter is bus agnostic, but wherever bus specifics are necessary, it assumes PCI. To give you a flavor of other network technologies, we also touch on *Asynchronous Transfer Mode* (ATM). We end the chapter by pondering on performance and throughput.

NIC drivers are different from other driver classes in that they do not rely on */dev* or */sys* to communicate with user space. Rather, applications interact with a NIC driver via a network interface (for example, eth0 for the first Ethernet interface) that abstracts an underlying protocol stack.

Driver Data Structures

When you write a device driver for a NIC, you have to operate on three classes of data structures:

1. Structures that form the building blocks of the network protocol stack. The socket buffer or `struct sk_buff` defined in *include/linux/sk_buff.h* is the key structure used by the kernel's TCP/IP stack.

2. Structures that define the interface between the NIC driver and the protocol stack. `struct net_device` defined in *include/linux/netdevice.h* is the core structure that constitutes this interface.

3. Structures related to the I/O bus. PCI and its derivatives are common buses used by today's NICs.

We take a detailed look at socket buffers and the `net_device` interface in the next two sections. We covered PCI data structures in Chapter 10, "Peripheral Component Interconnect," so we won't revisit them here.

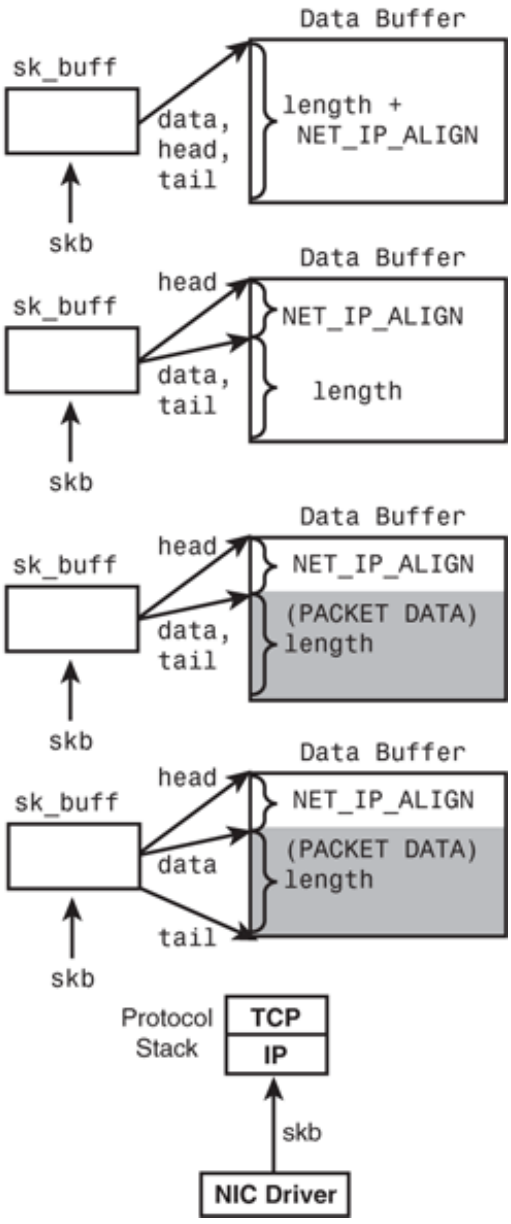
Socket Buffers

`sk_buff`s provide efficient buffer handling and flow-control mechanisms to Linux networking layers. Like DMA descriptors that contain metadata on DMA buffers, `sk_buff`s hold control information describing attached memory buffers that carry network packets (see Figure 15.1). `sk_buff`s are enormous structures having dozens of elements, but in this chapter we confine ourselves to those that interest the network device driver writer. An `sk_buff` links itself to its associated packet buffer using five main fields:

- `head`, which points to the start of the packet
- `data`, which points to the start of packet payload
- `tail`, which points to the end of packet payload
- `end`, which points to the end of the packet
- `len`, the amount of data that the packet contains

Figure 15.1. **`sk_buff`** operations.

```
struct sk_buff  *skb;
/* ... */

skb = dev_alloc_skb(length +
                    NET_IP_ALIGN);

skb_reserve(skb, NET_IP_ALIGN);

mempcy(skb->data, dma_buffer,
       length)

skb_put(skb, length);

netif_rx(skb);
```



Assume `skb` points to an `sk_buff`, `skb->head`, `skb->data`, `skb->tail`, and `skb->end` slide over the associated packet buffer as the packet traverses the protocol stack in either direction. `skb->data`, for example, points to the header of the protocol that is currently processing the packet. When a packet reaches the IP layer via the receive path, `skb->data` points to the IP header; when the packet passes on to TCP, however, `skb->data` moves to the start of the TCP header. And as the packet drives through various protocols adding or discarding header data, `skb->len` gets updated, too. `sk_buff`s also contain pointers other than the four major ones previously mentioned. `skb->nh`, for example, remembers the position of the network protocol header irrespective of the current position of `skb->data`.

To illustrate how a NIC driver works with `sk_buff`s, Figure 15.1 shows data transitions on the receive data path. For convenience of illustration, the figure simplistically assumes that the operations shown are executed in sequence. However, for operational efficiency in the real world, the first two steps (`dev_alloc_skb()` and

skb_reserve()) are performed while initially preallocating a ring of receive buffers; the third step is accomplished by the NIC hardware as it directly DMA's the received packet into a preallocated sk_buff; and the final two steps (skb_put() and netif_rx()) are executed from the receive interrupt handler.

To create an sk_buff to hold a received packet, Figure 15.1 uses dev_alloc_skb(). This is an interrupt-safe routine that allocates memory for an sk_buff and associates it with a packet payload buffer. dev_kfree_skb() accomplishes the reverse of dev_alloc_skb(). Figure 15.1 next calls skb_reserve() to add a 2-byte padding between the start of the packet buffer and the beginning of the payload. This starts the IP header at a performance-friendly 16-byte boundary because the preceding Ethernet headers are 14 bytes long. The rest of the code statements in Figure 15.1 fill the payload buffer with the received packet and move skb->data, skb->tail, and skb->len to reflect this operation.

There are more sk_buff access routines relevant to some NIC drivers. skb_clone(), for example, creates a copy of a supplied skb_buff without copying the contents of the associated packet buffer. Look inside *net/core/skbuff.c* for the full list of sk_buff library functions.

The Net Device Interface

NIC drivers use a standard interface to interact with the TCP/IP stack. The net_device structure, which is even more gigantic than the sk_buff structure, defines this communication interface. To prepare ourselves for exploring the innards of the net_device structure, let's first follow the steps traced by a NIC driver during initialization. Refer to init_mycard() in Listing 15.1 as we move along:

- The driver allocates a net_device structure using alloc_netdev(). More commonly, it uses a suitable wrapper around alloc_netdev(). An Ethernet NIC driver, for example, calls alloc_etherdev(). A WiFi driver (discussed in the next chapter) invokes alloc_ieee80211(), and an IrDa driver calls upon alloc_irdadev(). All these functions take the size of a private data area as argument and create this area in addition to the net_device itself:

```
struct net_device  *netdev;
struct priv_struct *mycard_priv;
netdev      = alloc_etherdev(sizeof(struct
                             priv_struct));
mycard_priv = netdev->priv; /* Private area created
                               by alloc_etherdev() */
```

- Next, the driver populates various fields in the net_device that it allocated and registers the populated net_device with the network layer using register_netdev(netdev).

- The driver reads the NIC's *Media Access Control* (MAC) address from an accompanying EEPROM and configures *Wake-On-LAN* (WOL) if required. Ethernet controllers usually have a companion nonvolatile EEPROM to hold information such as their MAC address and WOL pattern, as shown in Figure 15.2. The former is a unique 48-bit address that is globally assigned. The latter is a magic sequence; if found in received data, it rouses the NIC if it's in suspend mode.

- If the NIC needs on-card firmware to operate, the driver downloads it using request_firmware(), as discussed in the section "Microcode Download" in Chapter 4, "Laying the Groundwork."

Let's now look at the methods that define the net_device interface. We categorize them under six heads for simplicity. Wherever relevant, this section points you to the example NIC driver developed in Listing 15.1 of the section "Device Example: Ethernet NIC."

Activation

The net_device interface requires conventional methods such as open(), close(), and ioctl(). The kernel opens an interface when you activate it using a tool such as *ifconfig:*

```
bash> ifconfig eth0 up
```

open() sets up receive and transmit DMA descriptors and other driver data structures. It also registers the NIC's interrupt handler by calling request_irq(). The net_device structure is passed as the devid argument to request_irq() so that the interrupt handler gets direct access to the associated net_device. (See mycard_open() and mycard_interrupt() in Listing 15.1 to find out how this is done.)

The kernel calls close() when you pull down an active network interface. This accomplishes the reverse of open().

Data Transfer

Data transfer methods form the crux of the net_device interface. In the transmit path, the driver supplies a method called hard_start_xmit, which the protocol layer invokes to pass packets down for onward transmission:

Code View:
```
netdev->hard_start_xmit = &mycard_xmit_frame; /* Transmit Method. See Listing 15.1 */
```

Until recently, network drivers didn't provide a net_device method for collecting received data. Instead, they asynchronously interrupted the protocol layer with packet payload. This old interface has, however, given way to a *New API* (NAPI) that is a mixture of an interrupt-driven driver push and a poll-driver protocol pull. A NAPI-aware driver thus needs to supply a poll() method and an associated weight that controls polling fairness:

```
netdev->poll   = &mycard_poll; /* Poll Method. See Listing 15.1 */
netdev->weight = 64;
```

We elaborate on data-transfer methods in the section "Talking with Protocol Layers."

Watchdog

The net_device interface provides a hook to return an unresponsive NIC to operational state. If the protocol layer senses no transmissions for a predetermined amount of time, it assumes that the NIC has hung and invokes a driver-supplied recovery method to reset the card. The driver sets the watchdog timeout through netdev->watchdog_timeo and registers the address of the recovery function via netdev->tx_timeout:

```
netdev->tx_timeout = &mycard_timeout; /* Method to reset the NIC */
netdev->watchdog_timeo = 8*HZ;        /* Reset if no response
                                         detected for 8 seconds */
```

Because the recovery method executes in timer-interrupt context, it usually schedules a task outside of that context to reset the NIC.

Statistics

To enable user land to collect network statistics, the NIC driver populates a net_device_stats structure and

provides a `get_stats()` method to retrieve it. Essentially the driver does the following:

1. Updates different types of statistics from relevant entry points:

```
#include <linux/netdevice.h>
struct net_device_stats mycard_stats;

static irqreturn_t
mycard_interrupt(int irq, void *dev_id)
{
  /* ... */
  if (packet_received_without_errors) {
    mycard_stats.rx_packets++;   /* One more received
                                    packet */
  }
  /* ... */
}
```

2. Implements the `get_stats()` method to retrieve the statistics:

```
static struct net_device_stats
*mycard_get_stats(struct net_device *netdev)
{
   /* House keeping */
   /* ... */
   return(&mycard_stats);
}
```

3. Supplies the retrieve method to higher layers:

```
netdev->get_stats = &mycard_get_stats;
/* ... */
register_netdev(netdev);
```

To collect statistics from your NIC, trigger invocation of `mycard_get_stats()` by executing an appropriate user mode command. For example, to find the number of packets received through the eth0 interface, do this:

```
bash> cat /sys/class/net/eth0/statistics/rx_packets
124664
```

WiFi drivers need to track several parameters not relevant to conventional NICs, so they implement a statistic collection method called `get_wireless_stats()` in addition to `get_stats()`. The mechanism for registering `get_wireless_stats()` for the benefit of WiFi-aware user space utilities is discussed in the section "WiFi" in the next chapter.

Configuration

NIC drivers need to support user space tools that are responsible for setting and getting device parameters. *Ethtool* configures parameters for Ethernet NICs. To support ethtool, the underlying NIC driver does the following:

1. Populates an `ethtool_ops` structure, defined in *include/linux/ethtool.h* with prescribed entry points:

```
#include <linux/ethtool.h>

/* Ethtool_ops methods */
struct ethtool_ops mycard_ethtool_ops = {
  /* ... */
  .get_eeprom = mycard_get_eeprom, /* Dump EEPROM
                                      contents */
  /* ... */
};
```

2. Implements the methods that are part of `ethtool_ops`:

```
static int
mycard_get_eeprom(struct net_device *netdev,
                  struct ethtool_eeprom *eeprom,
                  uint8_t *bytes)
{
   /* Access the accompanying EEPROM and pull out data */
   /* ... */
}
```

3. Exports the address of its `ethtool_ops`:

```
netdev->ethtool_ops = &mycard_ethtool_ops;
/* ... */
register_netdev(netdev);
```

After these are done, ethtool can operate over your Ethernet NIC. To dump EEPROM contents using ethtool, do this:

```
bash> ethtool -e eth0
Offset          Values
------          ------
0x0000          00 0d 60 79 32 0a 00 0b ff ff 10 20 ff ff ff ff
...
```

Ethtool comes packaged with some distributions; but if you don't have it, download it from http://sourceforge.net/projects/gkernel/. Refer to the man page for its full capabilities.

There are more configuration-related methods that a NIC driver provides to higher layers. An example is the method to change the MTU size of the network interface. To support this, supply the relevant method to `net_device`:

```
netdev->change_mtu = &mycard_change_mtu;
/* ... */
register_netdev(netdev);
```

The kernel invokes `mycard_change_mtu()` when you execute a suitable user command to alter the MTU of your card:

```
bash> echo 1500 > /sys/class/net/eth0/mtu
```

Bus Specific

Next come bus-specific details such as the start address and size of the NIC's on-card memory. For a PCI NIC driver, this configuration will look like this:

```
netdev->mem_start = pci_resource_start(pdev, 0);
netdev->mem_end   = netdev->mem_start + pci_resource_len(pdev, 0);
```

We discussed PCI resource functions in Chapter 10.

Talking with Protocol Layers

In the preceding section, you discovered the driver methods demanded by the `net_device` interface. Let's now take a closer look at how network data flows over this interface.

Receive Path

You learned in Chapter 4 that softirqs are bottom half mechanisms used by performance-sensitive subsystems. NIC drivers use `NET_RX_SOFTIRQ` to offload the work of posting received data packets to protocol layers. The driver achieves this by calling `netif_rx()` from its receive interrupt handler:

```
netif_rx(skb);   /* struct sk_buff *skb */
```

NAPI, alluded to earlier, improves this conventional interrupt-driven receive algorithm to lower demands on CPU utilization. When network load is heavy, the system might get bogged down by the large number of interrupts that it takes. NAPI's strategy is to use a polled mode when network activity is heavy but fall back to interrupt mode when the traffic gets light. NAPI-aware drivers switch between interrupt and polled modes based on network load. This is done as follows:

1. In interrupt mode, the interrupt handler posts received packets to protocol layers by scheduling `NET_RX_SOFTIRQ`. It then disables NIC interrupts and switches to polled mode by adding the device to a poll list:

   ```
   if (netif_rx_schedule_prep(netdev)) /* Housekeeping */ {
     /* Disable NIC interrupt */
     disable_nic_interrupt();
     /* Post the packet to the protocol layer and
        add the device to the poll list */
     __netif_rx_schedule(netdev);
   }
   ```

2. The driver provides a `poll()` method via its `net_device` structure.

3. In the polled mode, the driver's `poll()` method processes packets in the ingress queue. When the queue becomes empty, the driver re-enables interrupts and switches back to interrupt mode by calling `netif_rx_complete()`.

Look at `mycard_interrupt()`, `init_mycard()`, and `mycard_poll()` in Listing 15.1 to see NAPI in action.

Transmit Path

For data transmission, the interaction between protocol layers and the NIC driver is straightforward. The protocol stack invokes the driver's `hard_start_xmit()` method with the outgoing `sk_buff` as argument. The driver gets the packet out of the door by DMA-ing packet data to the NIC. DMA and the management of related data structures for PCI NIC drivers were discussed in Chapter 10.

The driver programs the NIC to interrupt the processor after it finishes transmitting a predetermined number of packets. Only when a transmit-complete interrupt occurs signaling completion of a transmit operation can the

driver reclaim or free resources such as DMA descriptors, DMA buffers, and `sk_buffs` associated with the transmitted packet.

Flow Control

The driver conveys its readiness or reluctance to accept protocol data by, respectively, calling `netif_start_queue()` and `netif_stop_queue()`.

During device `open()`, the NIC driver calls `netif_start_queue()` to ask the protocol layer to start adding transmit packets to the egress queue. During normal operation, however, the driver might require egress queuing to stop on occasion. Examples include the time window when the driver is replenishing data structures, or when it's closing the device. Throttling the downstream flow is accomplished by calling `netif_stop_queue()`. To request the networking stack to restart egress queuing, say when there are sufficient free buffers, the NIC driver invokes `netif_wake_queue()`. To check the current flow-control state, toss a call to `netif_queue_stopped()`.

Buffer Management and Concurrency Control

A high-performance NIC driver is a complex piece of software requiring intricate data structure management. As discussed in the section "Data Transfer" in Chapter 10, a NIC driver maintains linked lists (or "rings") of transmit and receive DMA descriptors, and implements free and in-use pools for buffer management. The driver typically implements a multipronged strategy to maintain buffer levels: preallocate a ring of DMA descriptors and associated sk_buffs during device open, replenish free pools by allocating new memory if available buffers dip below a predetermined watermark, and reclaim used buffers into the free pool when the NIC generates transmit-complete and receive interrupts.

Each element in the NIC driver's receive ring, for example, is populated as follows:

```
/* Allocate an sk_buff and the associated data buffer.
   See Figure 15.1 */
skb = dev_alloc_skb(MAX_NIC_PACKET_SIZE);
/* Align the data pointer */
  skb_reserve(skb, NET_IP_ALIGN);
/* DMA map for NIC access. The following invocation assumes a PCI
   NIC. pdev is a pointer to the associated pci_dev structure */
pci_map_single(pdev, skb->data, MAX_NIC_PACKET_SIZE,
               PCI_DMA_FROMDEVICE);
/* Create a descriptor containing this sk_buff and add it
   to the RX ring */
/* ... */
```

During reception, the NIC directly DMA's data to an sk_buff in the preceding preallocated ring and interrupts the processor. The receive interrupt handler, in turn, passes the packet to higher protocol layers. Developing ring data structures will make this discussion as well as the example driver in the next section loaded, so refer to the sources of the Intel PRO/1000 driver in the *drivers/net/e1000/* directory for a complete illustration.

Concurrent access protection goes hand-in-hand with managing such complex data structures in the face of multiple execution threads such as transmit, receive, transmit-complete interrupts, receive interrupts, and NAPI polling. We discussed several concurrency control techniques in Chapter 2, "A Peek Inside the Kernel."

Device Example: Ethernet NIC

Now that you have the background, it's time to write a NIC driver by gluing the pieces discussed so far. Listing 15.1 implements a skeletal Ethernet NIC driver. It only implements the main `net_device` methods. For help in developing the rest of the methods, refer to the *e1000* driver mentioned earlier. Listing 15.1 is generally independent of the underlying I/O bus but is slightly tilted to PCI. If you are writing a PCI NIC driver, you have to blend Listing 15.1 with the example PCI driver implemented in Chapter 10.

Listing 15.1. An Ethernet NIC Driver

Code View:

```
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/skbuff.h>
#include <linux/ethtool.h>

struct net_device_stats mycard_stats;   /* Statistics */

/* Fill ethtool_ops methods from a suitable place in the driver */
struct ethtool_ops mycard_ethtool_ops = {
  /* ... */
  .get_eeprom = mycard_get_eeprom,      /* Dump EEPROM contents */
  /* ... */
};

/* Initialize/probe the card. For PCI cards, this is invoked
   from (or is itself) the probe() method. In that case, the
   function is declared as:
   static struct net_device *init_mycard(struct pci_dev *pdev, const
                                         struct pci_device_id *id)
*/
static struct net_device *
init_mycard()
{
  struct net_device *netdev;
  struct priv_struct mycard_priv;

   /* ... */
  netdev = alloc_etherdev(sizeof(struct priv_struct));
  /* Common methods */
  netdev->open = &mycard_open;
  netdev->stop = &mycard_close;
  netdev->do_ioctl = &mycard_ioctl;

  /* Data transfer */
  netdev->hard_start_xmit = &mycard_xmit_frame; /* Transmit */
  netdev->poll = &mycard_poll;                  /* Receive - NAPI  */
  netdev->weight = 64;                          /* Fairness */

  /* Watchdog */
  netdev->tx_timeout = &mycard_timeout;      /* Recovery function */
  netdev->watchdog_timeo = 8*HZ;             /* 8-second timeout */

  /* Statistics and configuration */
  netdev->get_stats = &mycard_get_stats;     /* Statistics support */
  netdev->ethtool_ops = &mycard_ethtool_ops; /* Ethtool support */
```

```
  netdev->set_mac_address = &mycard_set_mac; /* Change MAC */
  netdev->change_mtu = &mycard_change_mtu;   /* Alter MTU */

  strncpy(netdev->name, pci_name(pdev),
          sizeof(netdev->name) - 1);         /* Name (for PCI) */

  /* Bus-specific parameters. For a PCI NIC, it looks as follows */
  netdev->mem_start = pci_resource_start(pdev, 0);
  netdev->mem_end   = netdev->mem_start + pci_resource_len(pdev, 0);

  /* Register the interface */
  register_netdev(netdev);

  /* ... */

  /* Get MAC address from attached EEPROM */
  /* ... */

  /* Download microcode if needed */
  /* ... */
}


/* The interrupt handler */
static irqreturn_t
mycard_interrupt(int irq, void *dev_id)
{
  struct net_device *netdev = dev_id;
  struct sk_buff *skb;
  unsigned int length;

  /* ... */

  if (receive_interrupt) {
    /* We were interrupted due to packet reception. At this point,
       the NIC has already DMA'ed received data to an sk_buff that
       was pre-allocated and mapped during device open. Obtain the
       address of the sk_buff depending on your data structure
       design and assign it to 'skb'. 'length' is similarly obtained
       from the NIC by reading the descriptor used to DMA data from
       the card. Now, skb->data contains the receive data. */
    /* ... */

    /* For PCI cards, perform a pci_unmap_single() on the
       received buffer in order to allow the CPU to access it */
    /* ... */

    /* Allow the data go to the tail of the packet by moving
       skb->tail down by length bytes and increasing
       skb->len correspondingly */
    skb_put(skb, length)

    /* Pass the packet to the TCP/IP stack */
#if !defined (USE_NAPI)  /* Do it the old way */
    netif_rx(skb);
#else                    /* Do it the NAPI way */
    if (netif_rx_schedule_prep(netdev))) {
      /* Disable NIC interrupt. Implementation not shown. */
      disable_nic_interrupt();
```

```
      /* Post the packet to the protocol layer and
         add the device to the poll list */
      __netif_rx_schedule(netdev);
    }
#endif
  } else if (tx_complete_interrupt) {
    /* Transmit Complete Interrupt */
    /* ... */
    /* Unmap and free transmit resources such as
       DMA descriptors and buffers. Free sk_buffs or
       reclaim them into a free pool */
    /* ... */

 }
}

/* Driver open */
static int
mycard_open(struct net_device *netdev)
{
  /* ... */

  /* Request irq */
  request_irq(irq, mycard_interrupt, IRQF_SHARED,
              netdev->name, dev);

  /* Fill transmit and receive rings */
  /* See the section,
     "Buffer Management and Concurrency Control" */
  /* ... */

  /* Provide free descriptor addresses to the card */
  /* ... */

  /* Convey your readiness to accept data from the
     networking stack */
   netif_start_queue(netdev);

  /* ... */
}

/* Driver close */
static int
mycard_close(struct net_device *netdev)
{
  /* ... */

  /* Ask the networking stack to stop sending down data */
   netif_stop_queue(netdev);

  /* ... */
}
/* Called when the device is unplugged or when the module is
   released. For PCI cards, this is invoked from (or is itself)
   the remove() method. In that case, the function is declared as:
   static void __devexit mycard_remove(struct pci_dev *pdev)
*/
static void __devexit
mycard_remove()
```

```
{
  struct net_device *netdev;

  /* ... */
  /* For a PCI card, obtain the associated netdev as follows,
     assuming that the probe() method performed a corresponding
     pci_set_drvdata(pdev, netdev) after allocating the netdev */
  netdev = pci_get_drvdata(pdev); /*

  unregister_netdev(netdev);  /* Reverse of register_netdev() */

  /* ... */

  free_netdev(netdev);       /* Reverse of alloc_netdev() */

  /* ... */
}


/* Suspend method. For PCI devices, this is part of
   the pci_driver structure discussed in Chapter 10 */
static int
mycard_suspend(struct pci_dev *pdev, pm_message_t state)
{
  /* ... */
  netif_device_detach(netdev);
  /* ... */
}


/* Resume method. For PCI devices, this is part of
   the pci_driver structure discussed in Chapter 10 */
static int
mycard_resume(struct pci_dev *pdev)
{
  /* ... */
  netif_device_attach(netdev);
  /* ... */
}


/* Get statistics */
static struct net_device_stats *
mycard_get_stats(struct net_device *netdev)
{
  /* House keeping */
  /* ... */

  return(&mycard_stats);
}

/* Dump EEPROM contents. This is an ethtool_ops operation */
static int
mycard_get_eeprom(struct net_device *netdev,
                  struct ethtool_eeprom *eeprom, uint8_t *bytes)
{
  /* Read data from the accompanying EEPROM */
  /* ... */
}
```

```
/* Poll method */
static int
mycard_poll(struct net_device *netdev, int *budget)
{
   /* Post packets to the protocol layer using
      netif_receive_skb() */
   /* ... */

   if (no_more_ingress_packets()){
     /* Remove the device from the polled list */
     netif_rx_complete(netdev);

     /* Fall back to interrupt mode. Implementation not shown */
     enable_nic_interrupt();

     return 0;
  }
}
/* Transmit method */
static int
mycard_xmit_frame(struct sk_buff *skb, struct net_device *netdev)
{
  /* DMA the transmit packet from the associated sk_buff
     to card memory */
  /* ... */
  /* Manage buffers */
  /* ... */
}
```

Ethernet PHY

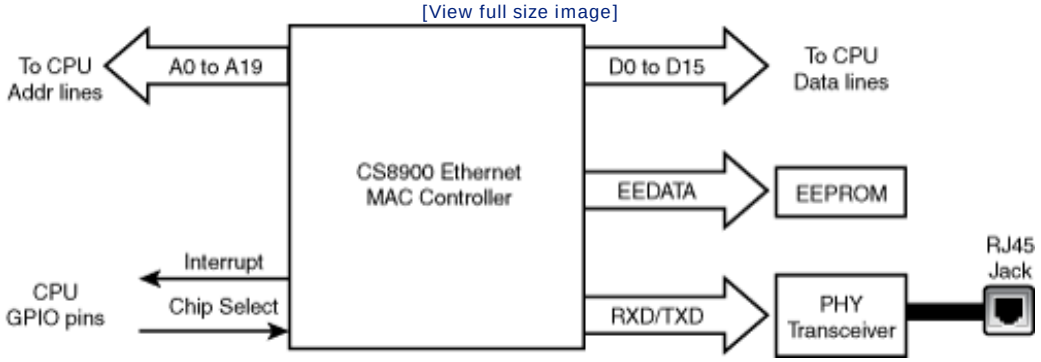
Ethernet controllers implement the MAC layer and have to be used in tandem with a Physical layer (PHY) transceiver. The former corresponds to the datalink layer of the Open Systems Interconnect (OSI) model, while the latter implements the physical layer. Several SoCs have built-in MACs that connect to external PHYs. The Media Independent Interface (MII) is a standard interface that connects a Fast Ethernet MAC to a PHY. The Ethernet device driver communicates with the PHY over MII to configure parameters such as PHY ID, line speed, duplex mode, and auto negotiation. Look at *include/linux/mii.h* for MII register definitions.

ISA Network Drivers

Let's now take a peek at an ISA NIC. The CS8900 is a 10Mbps Ethernet controller chip from Crystal Semiconductor (now Cirrus Logic). This chip is commonly used to Ethernet-enable embedded devices, especially for debug purposes. Figure 15.2 shows a connection diagram surrounding a CS8900. Depending on the processor on your board and the chip-select used to drive the chip, the CS8900 registers map to different regions in the CPU's I/O address space. The device driver for this controller is an ISA-type driver (look at the section "ISA and MCA" in Chapter 20, "More Devices and Drivers") that probes candidate address regions to detect the controller's presence. The ISA probe method elicits the controller's I/O base address by looking for a signature such as the chip ID.

Figure 15.2. Connection diagram surrounding a CS8900 Ethernet controller.

[View full size image]



Look at *drivers/net/cs89x0.c* for the source code of the CS8900 driver. `cs89x0_probe1()` probes I/O address ranges to sense a CS8900. It then reads the current configuration of the chip. During this step, it accesses the CS8900's companion EEPROM and gleans the controller's MAC address. Like the driver in Listing 15.1, *cs89x0.c* is also built using `netif_*()` and `skb_*()` interface routines.

Some platforms that use the CS8900 allow DMA. ISA devices, unlike PCI cards, do not have DMA mastering capabilities, so they need an external DMA controller to transfer data.

Asynchronous Transfer Mode

ATM is a high-speed, connection-oriented, back-bone technology. ATM guarantees high *quality of service* (QoS) and low latencies, so it's used for carrying audio and video traffic in addition to data.

Here's a quick summary of the ATM protocol: ATM communication takes place in units of 53-byte cells. Each cell begins with a 5-byte header that carries a *virtual path identifier* (VPI) and a *virtual circuit identifier* (VCI). ATM connections are either *switched virtual circuits* (SVCs) or *permanent virtual circuits* (PVCs). During SVC establishment, VPI/VCI pairs are configured in intervening ATM switches to route incoming cells to appropriate egress ports. For PVCs, the VPI/VCI pairs are permanently configured in the ATM switches and not set up and torn down for each connection.

There are three ways you can run TCP/IP over ATM, all of which are supported by Linux-ATM:

1. Classical IP over ATM (CLIP) as specified in RFC[1] 1577.

   [1] *Request For Comments* (RFC) are documents that specify networking standards.

2. Emulating a LAN over an ATM network. This is called *LAN Emulation* (LANE).

3. *Multi Protocol over ATM* (MPoA). This is a routing technique that improves performance.

Linux-ATM is an experimental collection of kernel drivers, user space utilities, and daemons. You will find ATM drivers and protocols under *drivers/atm/* and *net/atm/*, respectively, in the source tree. http://linux-atm.sourceforge.net/ hosts user-space programs required to use Linux-ATM. Linux also incorporates an ATM socket API consisting of SVC sockets (AF_ATMSVC) and PVC sockets (AF_ATMPVC).

A protocol called *Multiprotocol Label Switching* (MPLS) is replacing ATM. The Linux-MPLS project, hosted at http://mpls-linux.sourceforge.net/, is not yet part of the mainline kernel.

We look at some ATM-related throughput issues in the next section.

Network Throughput

Several tools are available to benchmark network performance. *Netperf*, available for free from www.netperf.org, can set up complex TCP/UDP connection scenarios. You can use scripts to control characteristics such as protocol parameters, number of simultaneous sessions, and size of data blocks. Benchmarking is accomplished by comparing the resulting throughput with the maximum practical bandwidth that the networking technology yields. For example, a 155Mbps ATM adapter produces a maximum IP throughput of 135Mbps, taking into account the ATM cell header size, overheads due to the *ATM Adaptation Layer* (AAL), and the occasional maintenance cells sent by the physical *Synchronous Optical Networking* (SONET) layer.

To obtain optimal throughput, you have to design your NIC driver for high performance. In addition, you need an in-depth understanding of the network protocol that your driver ferries.

Driver Performance

Let's take a look at some driver design issues that can affect the horsepower of your NIC:

- Minimizing the number of instructions in the main data path is a key criterion while designing drivers for fast NICs. Consider a 1Gbps Ethernet adapter with 1MB of on-board memory. At line rate, the card memory can hold up to 8 milliseconds of received data. This directly translates to the maximum allowable instruction path length. Within this path length, incoming packets have to be reassembled, DMAed to memory, processed by the driver, protected from concurrent access, and delivered to higher layer protocols.

- During *programmed I/O* (PIO), data travels all the way from the device to the CPU, before it gets written to memory. Moreover, the CPU gets interrupted whenever the device needs to transfer data, and this contributes to latencies and context switch delays. DMAs do not suffer from these bottlenecks, but can turn out to be more expensive than PIOs if the data to be transferred is less than a threshold. This is because small DMAs have high relative overheads for building descriptors and flushing corresponding processor cache lines for data coherency. A performance-sensitive device driver might use PIO for small packets and DMA for larger ones, after experimentally determining the threshold.

- For PCI network cards having DMA mastering capability, you have to determine the optimal DMA burst size, which is the time for which the card controls the bus at one stretch. If the card bursts for a long duration, it may hog the bus and prevent the processor from keeping up with data DMA-ed previously. PCI drivers program the burst size via a register in the PCI configuration space. Normally the NIC's burst size is programmed to be the same as the cache line size of the processor, which is the number of bytes that the processor reads from system memory each time there is a cache miss. In practice, however, you might need to connect a bus analyzer to determine the beneficial burst duration because factors such as the presence of a split bus (multiple bus types like ISA and PCI) on your system can influence the optimal value.

- Many high-speed NICs offer the capability to offload the CPU-intensive computation of TCP checksums from the protocol stack. Some support DMA scatter-gather that we visited in Chapter 10. The driver needs to leverage these capabilities to achieve the maximum practical bandwidth that the underlying network yields.

- Sometimes, a driver optimization might create unexpected speed bumps if it's not sensitive to the implementation details of higher protocols. Consider an NFS-mounted filesystem on a computer equipped

with a high-speed NIC. Assume that the NIC driver takes only occasional transmit complete interrupts to minimize latencies, but that the NFS server implementation uses freeing of its transmit buffers as a flow-control mechanism. Because the driver frees NFS transmit buffers only during the sparsely generated transmit complete interrupts, file copies over NFS crawl, even as Internet downloads zip along yielding maximum throughput.

Protocol Performance

Let's now dig into some protocol-specific characteristics that can boost or hurt network throughput:

- TCP window size can impact throughput. The window size provides a measure of the amount of data that can be transmitted before receiving an acknowledgment. For fast NICs, a small window size might result in TCP sitting idle, waiting for acknowledgments of packets already transmitted. Even with a large window size, a small number of lost TCP packets can affect performance because lost frames can use up the window at line speeds. In the case of UDP, the window size is not relevant because it does not support acknowledgments. However, a small packet loss can spiral into a big rate drop due to the absence of flow-control mechanisms.

- As the block size of application data written to TCP sockets increases, the number of buffers copied from user space to kernel space decreases. This lowers the demand on processor utilization and is good for performance. If the block size crosses the MTU corresponding to the network protocol, however, processor cycles get wasted on fragmentation. The desirable block size is thus the outgoing interface MTU, or the largest packet that can be sent without fragmentation through an IP path if Path MTU discovery mechanisms are in operation. While running IP over ATM, for example, because the ATM adaptation layer has a 64K MTU, there is virtually no upper bound on block size. (RFC 1626 defaults this to 9180.) If you are running IP over ATM LANE, however, the block size should mirror the MTU size of the respective LAN technology being emulated. It should thus be 1500 for standard Ethernet, 8000 for jumbo Gigabit Ethernet, and 18K for 16Mbps Token Ring.

Looking at the Sources

The *drivers/net/* directory contains sources of various NIC drivers. Look inside *drivers/net/e1000/* for an example NIC driver. You will find network protocol implementations in the *net/* directory. `sk_buff` access routines are in *net/core/skbuff.c*. Library routines that aid the implementation of your driver's `net_device` interface stay in *net/core/dev.c* and *include/linux/netdevice.h*.

TUN/TAP Driver

The TUN/TAP device driver *drivers/net/tun.c*, used for protocol tunneling, is an example of a combination of a virtual network driver and a pseudo char driver. The pseudo char device (*/dev/net/tun*) acts as the underlying hardware for the virtual network interface (*tunX*), so instead of transmitting frames to a physical network, the TUN network driver sends it to an application that is reading from */dev/net/tun*. Similarly, instead of receiving data from a physical network, the TUN driver accepts it from an application writing to */dev/net/tun*. Look at *Documentation/networking/tuntap.txt* for more explanations and usage scenarios. Since both network and char portions of the driver do not have to deal with the complexities of hardware interaction, it serves as a very readable, albeit simplistic, driver example.

Files under */sys/class/net/* let you operate on NIC driver parameters. Use the nodes under */proc/sys/net/* to configure protocol-specific variables. To set the maximum TCP transmit window size, for example, echo the desired value to */proc/sys/net/core/wmem_max*. The */proc/net/* directory has a collection of system-specific network information. Examine */proc/net/dev* for statistics on all NICs on your system and look at */proc/net/arp* for the ARP table.

Table 15.1 contains the main data structures used in this chapter and their location in the source tree. Table 15.2 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 15.1. Summary of Data Structures

Data Structure	Location	Description
`sk_buff`	*include/linux/skbuff.h*	`sk_buff`s provide efficient buffer handling and flow-control mechanisms to Linux networking layers.
`net_device`	*include/linux/netdevice.h*	Interface between NIC drivers and the TCP/IP stack.
`net_device_stats`	*include/linux/netdevice.h*	Statistics pertaining to a network device.
`ethtool_ops`	*include/linux/ethtool.h*	Entry points to tie a NIC driver to the ethtool utility.

Table 15.2. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`alloc_netdev()`	*net/core/dev.c*	Allocates a `net_device`

Kernel Interface	Location	Description
alloc_etherdev() alloc_ieee80211() alloc_irdadev()	*net/ethernet/eth.c* *net/ieee80211/ieee80211_module.c* *net/irda/irda_device.c*	Wrappers to alloc_netdev()
free_netdev()	*net/core/dev.c*	Reverse of alloc_netdev()
register_netdev()	*net/core/dev.c*	Registers a net_device
unregister_netdev()	*net/core/dev.c*	Unregisters a net_device
dev_alloc_skb()	*include/linux/skbuff.h*	Allocates memory for an sk_buff and associates it with a packet payload buffer
dev_kfree_skb()	*include/linux/skbuff.h* *net/core/skbuff.c*	Reverse of dev_alloc_skb()
skb_reserve()	*include/linux/skbuff.h*	Adds a padding between the start of a packet buffer and the beginning of payload
skb_clone()	*net/core/skbuff.c*	Creates a copy of a supplied sk_buff without copying the contents of the associated packet buffer
skb_put()	*include/linux/skbuff.h*	Allows packet data to go to the tail of the packet
netif_rx()	*net/core/dev.c*	Passes a network packet to the TCP/IP stack
netif_rx_schedule_prep() __netif_rx_schedule()	*include/linux/netdevice.h* *net/core/dev.c*	Passes a network packet to the TCP/IP stack (NAPI)
netif_receive_skb()	*net/core/dev.c*	Posts packet to the protocol layer from the poll() method (NAPI)
netif_rx_complete()	*include/linux/netdevice.h*	Removes a device from polled list (NAPI)
netif_device_detach()	*net/core/dev.c*	Detaches the device (commonly called during power suspend)
netif_device_attach()	*net/core/dev.c*	Attaches the device (commonly called during power resume)
netif_start_queue()	*include/linux/netdevice.h*	Conveys readiness to accept data from the networking stack
netif_stop_queue()	*include/linux/netdevice.h*	Asks the networking stack to stop sending down data
netif_wake_queue()	*include/linux/netdevice.h*	Restarts egress queuing
netif_queue_stopped()	*include/linux/netdevice.h*	Checks flow-control state

Chapter 16. Linux Without Wires

In This Chapter

- Bluetooth

467

- Infrared

478

- WiFi

489

- Cellular Networking

496

- Current Trends

500

Several small-footprint devices are powered by the dual combination of a wireless technology and Linux. Bluetooth, Infrared, WiFi, and cellular networking are established wireless technologies that have healthy Linux support. Bluetooth eliminates cables, injects intelligence into dumb devices, and opens a flood gate of novel applications. Infrared is a low-cost, low-range, medium-rate, wireless technology that can network laptops, connect handhelds, or dispatch a document to a printer. WiFi is the wireless equivalent of an Ethernet LAN. Cellular networking using *General Packet Radio Service* (GPRS) or *code division multiple access* (CDMA) keeps you Internet-enabled on the go, as long as your wanderings are confined to service provider coverage area.

Because these wireless technologies are widely available in popular form factors, you are likely to end up, sooner rather than later, with a card that does not work on Linux right away. Before you start working on enabling an unsupported card, you need to know in detail how the kernel implements support for the corresponding technology. In this chapter, let's learn how Linux enables Bluetooth, Infrared, WiFi, and cellular networking.

Wireless Trade-Offs

Bluetooth, Infrared, WiFi, and GPRS serve different niches. The trade-offs can be gauged in terms of speed, range, cost, power consumption, ease of hardware/software co-design, and PCB real estate usage.

Table 16.1 gives you an idea of these parameters, but you will have to contend with several variables when you measure the numbers on the ground. The speeds listed are the theoretical maximums. The power consumptions indicated are relative, but in the real world they also depend on the vendor's implementation techniques, the technology subclass, and the operating mode. Cost economics depend on the chip form factor and whether the chip contains built-in microcode that implements some of the protocol layers. The board real estate consumed depends not just on the chipset, but also on transceivers, antennae, and whether you build using *off-the-shelf* (OTS) modules.

Bluetooth
720Kbps
10m to 100m
**
**
**
**
Infrared Data
4Mbps (Fast IR)
Up to 1 meter within a 30-degree cone
*
*
*
*
WiFi
54Mbps
150 meters (indoors)
****
***
***
***
GPRS
170Kbps
Service provider coverage
***
****
*
***

Note: The last four columns give relative measurement (depending on the number of * symbols) rather than absolute values.

Table 16.1. Wireless Trade-Offs

	Speed	Range	Power	Cost	Co-Design Effort	Board Real Estate

Some sections in this chapter focus more on "system programming" than device drivers. This is because the corresponding regions of the protocol stack (for example, Bluetooth RFCOMM and Infrared networking) are already present in the kernel and you are more likely to perform associated user mode customizations than develop protocol content or device drivers.

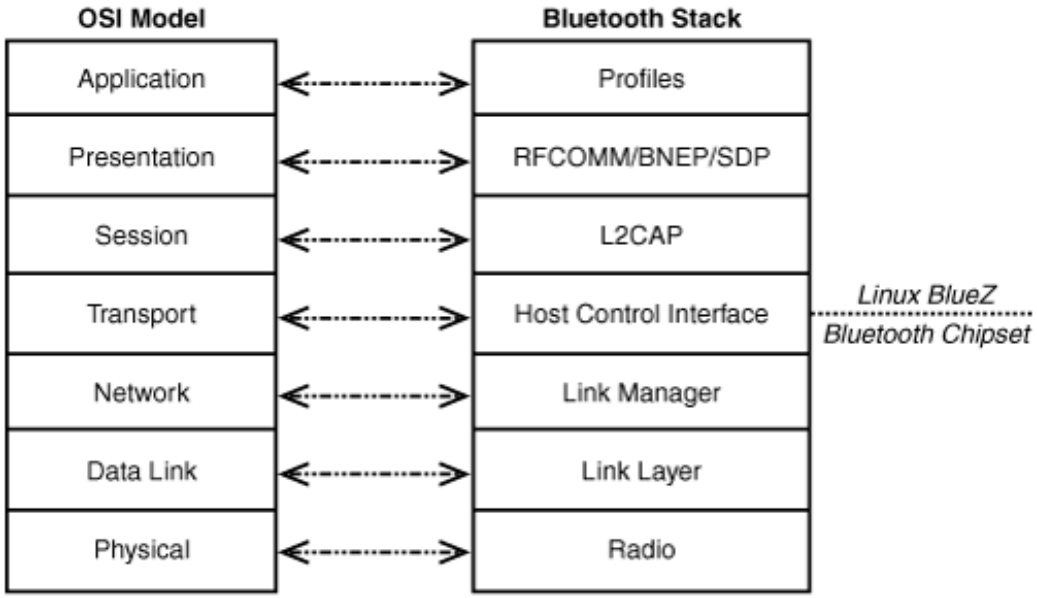
Bluetooth

Bluetooth is a short-range cable-replacement technology that carries both data and voice. It supports speeds of up to 723Kbps (asymmetric) and 432Kbps (symmetric). *Class 3* Bluetooth devices have a range of 10 meters, and *Class 1* transmitters can communicate up to 100 meters.

Bluetooth is designed to do away with wires that constrict and clutter your environment. It can, for example, turn your wristwatch into a front-end for a bulky *Global Positioning System* (GPS) hidden inside your backpack. Or it can, for instance, let you navigate a presentation via your handheld. Again, Bluetooth can be the answer if you want your laptop to be a hub that can Internet-enable your Bluetooth-aware MP3 player. If your wristwatch, handheld, laptop, or MP3 player is running Linux, knowledge of the innards of the Linux Bluetooth stack will help you extract maximum mileage out of your device.

As per the Bluetooth specification, the protocol stack consists of the layers shown in Figure 16.1 . The radio, link controller, and link manager roughly correspond to the physical, data link, and network layers in the *Open Systems Interconnect* (OSI) standard reference model. The *Host Control Interface* (HCI) is the protocol that carries data to/from the hardware and, hence, maps to the transport layer. The Bluetooth *Logical Link Control and Adaptation Protocol* (L2CAP) falls in the session layer. Serial port emulation using *Radio Frequency Communication* (RFCOMM), Ethernet emulation using *Bluetooth Network Encapsulation Protocol* (BNEP), and the *Service Discovery Protocol* (SDP) are part of the feature-rich presentation layer. At the top of the stack reside various application environments called *profiles.* The radio, link controller, and link manager are usually part of Bluetooth hardware, so operating system support starts at the HCI layer.

Figure 16.1. The Bluetooth stack.



A common method of interfacing Bluetooth hardware with a microcontroller is by connecting the chipset's data lines to the controller's UART pins. Figure 13.4 of Chapter 13 , "Audio Drivers," shows a Bluetooth chip on an MP3 player communicating with the processor via a UART. USB is another oft-used vehicle for communicating with Bluetooth chipsets. Figure 11.2 of Chapter 11 , "Universal Serial Bus," shows a Bluetooth chip on an embedded device interfacing with the processor over USB. Irrespective of whether you use UART or USB (we will look at both kinds of devices later), the packet format used to transport Bluetooth data is HCI.

BlueZ

The BlueZ Bluetooth implementation is part of the mainline kernel and is the official Linux Bluetooth stack.

Figure 16.2 shows how BlueZ maps Bluetooth protocol layers to kernel modules, kernel threads, user space daemons, configuration tools, utilities, and libraries. The main BlueZ components are explained here:

1. *bluetooth.ko* contains the core BlueZ infrastructure. All other BlueZ modules utilize its services. It's also responsible for exporting the Bluetooth family of sockets (`AF_BLUETOOTH`) to user space and for populating related sysfs entries.

2. For transporting Bluetooth HCI packets over UART, the corresponding BlueZ HCI implementation is *hci_uart.ko.* For USB transport, it's *hci_usb.ko* .

3. *l2cap.ko* implements the L2CAP adaptation layer that is responsible for segmentation and reassembly. It also multiplexes between different higher-layer protocols.

4. To run TCP/IP applications over Bluetooth, you have to emulate Ethernet ports over L2CAP using BNEP. This is accomplished by *bnep.ko.* To service BNEP connections, BlueZ spawns a kernel thread called *kbnepd* .

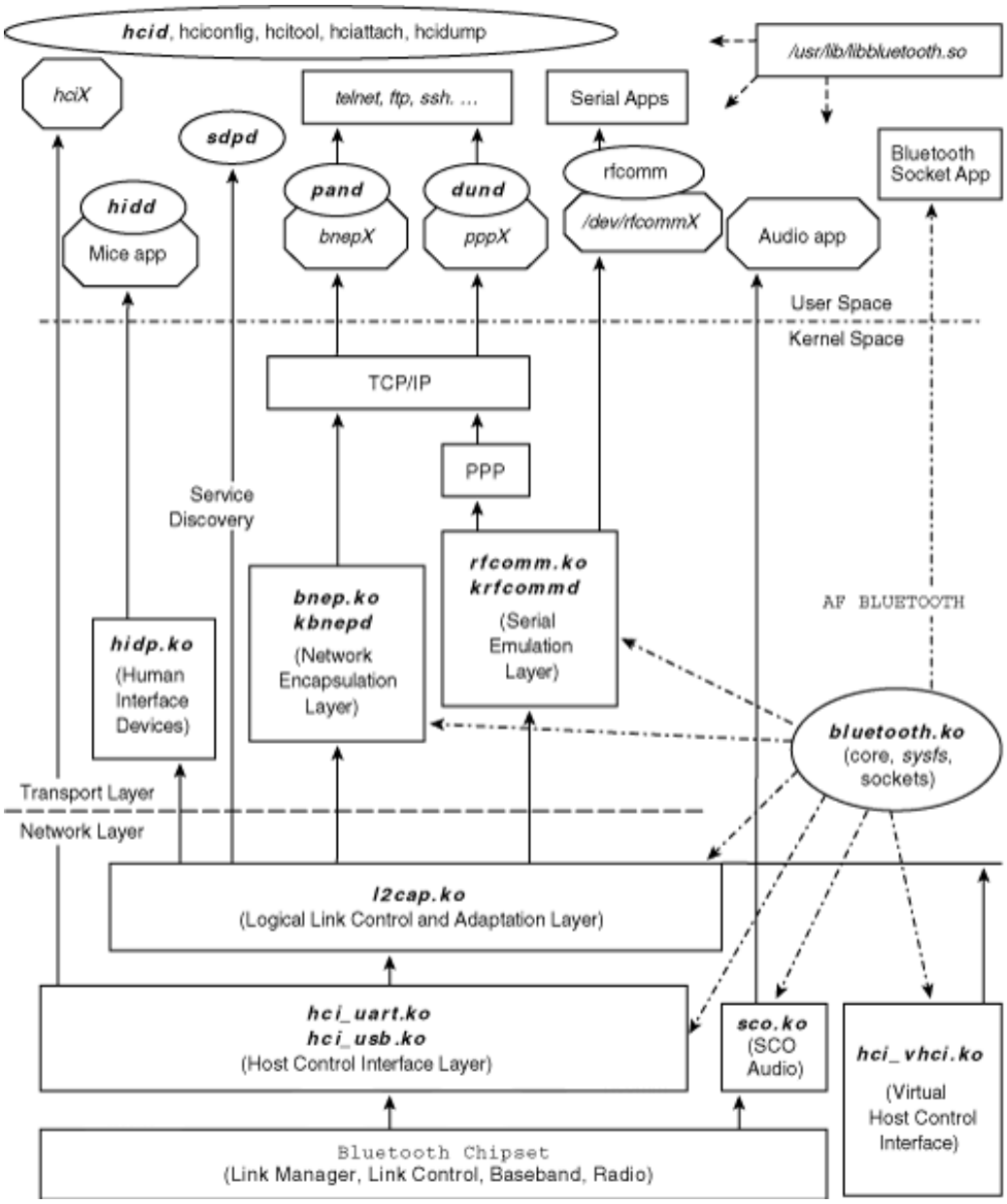
5. To run serial port applications such as terminal emulators over Bluetooth, you need to emulate serial ports over L2CAP. This is accomplished by *rfcomm.ko.* RFCOMM also functions as the pillar that supports networking over PPP. To service incoming RFCOMM connections, *rfcomm.ko* spawns a kernel thread called *krfcommd.* To set up and maintain connections to individual RFCOMM channels on target devices, use the *rfcomm* utility.

6. The *Human Interface Devices* (HID) layer is implemented via *hidp.ko* . The user mode daemon, *hidd* , lets BlueZ handle input devices such as Bluetooth mice.

7. Audio is handled via the *Synchronous Connection Oriented* (SCO) layer implemented by *sco.ko* .

Figure 16.2. Bluetooth protocol layers mapped to BlueZ kernel modules.

[View full size image]



Let's now trace the kernel code flow for two example Bluetooth devices: a *Compact Flash* (CF) card and a USB adapter.

Device Example: CF Card

The Sharp Bluetooth Compact Flash card is built using a Silicon Wave chipset and uses a serial transport to carry HCI packets. There are three different ways by which HCI packets can be transported serially:

1. H4 (UART), which is used by the Sharp CF card. H4 is the standard method to transfer Bluetooth data over UARTs as defined by the Bluetooth specification. Look at *drivers/bluetooth/hci_h4.c* for the BlueZ implementation.

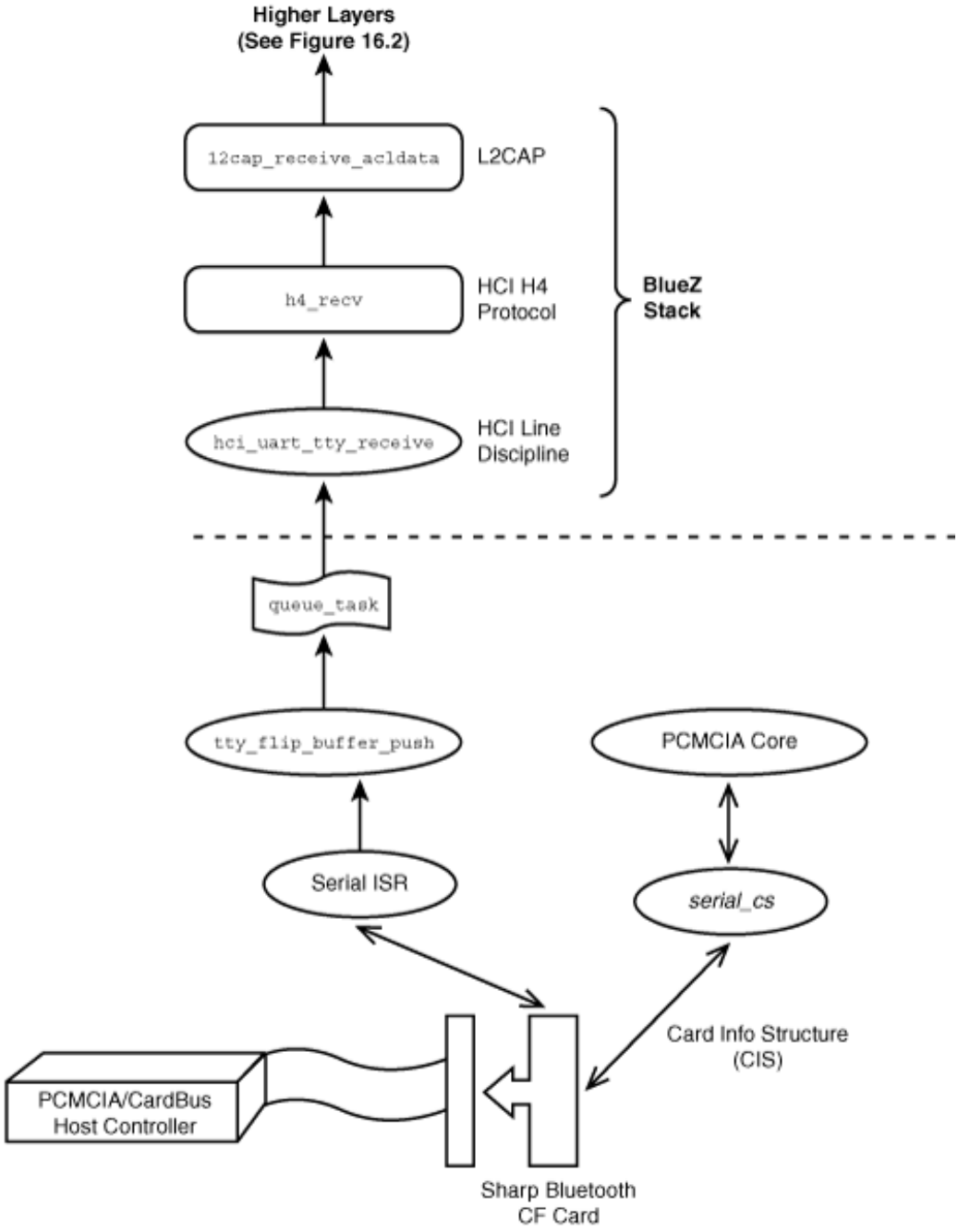
2. H3 (RS232). Devices using H3 are hard to find. BlueZ has no support for H3.

3. *BlueCore Serial Protocol* (BCSP), which is a proprietary protocol from *Cambridge Silicon Radio* (CSR) that supports error checking and retransmission. BCSP is used on non-USB devices based on CSR BlueCore chips including PCMCIA and CF cards. The BlueZ BCSP implementation lives in *drivers/-bluetooth/hci_bcsp.c* .

The read data path for the Sharp Bluetooth card is shown in Figure 16.3 . The first point of contact between the card and the kernel is at the UART driver. As you saw in Figure 9.5 of Chapter 9 , "PCMCIA and Compact Flash," the serial Card Services driver, *drivers/serial/serial_cs.c* , allows the rest of the operating system to see the Sharp card as if it were a serial device. The serial driver passes on the received HCI packets to BlueZ. BlueZ implements HCI processing in the form of a kernel line discipline. As you learned in Chapter 6 , "Serial Drivers," line disciplines reside above the serial driver and shape its behavior. The HCI line discipline invokes associated protocol routines (H4 in this case) for assistance in data processing. From then on, L2CAP and higher BlueZ layers take charge.

Figure 16.3. Read data path from a Sharp Bluetooth CF card.

[View full size image]



Device Example: USB Adapter

Let's now look at a device that uses USB to transport HCI packets. The Belkin Bluetooth USB adapter is one such gadget. In this case, the Linux USB layer (*drivers/usb/**), the HCI USB transport driver (*drivers/bluetooth/hci_usb.c*), and the BlueZ protocol stack (*net/bluetooth/**) are the main players that get the data rolling. Let's see how these three kernel layers interact.

As you learned in Chapter 11 , USB devices exchange data using one or more of four pipes. For Bluetooth USB devices, each pipe is responsible for carrying a particular type of data:

1. Control pipes are used to transport HCI commands.

2. Interrupt pipes are responsible for carrying HCI events.

3. Bulk pipes transfer *asynchronous connectionless* (ACL) Bluetooth data.

4. Isochronous pipes carry SCO audio data.

You also saw in Chapter 11 that when a USB device is plugged into a system, the host controller driver enumerates it using a control pipe and assigns endpoint addresses between 1 and 127. The configuration descriptor read by the USB subsystem during enumeration contains information about the device, such as its `class` , `subclass` , and `protocol` . The Bluetooth specification defines the (`class` , `subclass` , `protocol`) codes of Bluetooth USB devices as (`0xE, 0x01, 0x01`) . The HCI USB transport driver (*hci_usb*) registers these values with the USB core during initialization. When the Belkin USB adapter is plugged in, the USB core reads the (`class` , `subclass` , `protocol`) information from the device configuration descriptor. Because this information matches the values registered by *hci_usb* , this driver gets attached to the Belkin USB adapter. *hci_usb* reads Bluetooth data from the four USB pipes described previously and passes it on to the BlueZ protocol stack. Linux applications now run seamlessly over this device, as shown in Figure 16.2 .

RFCOMM

RFCOMM emulates serial ports over Bluetooth. Applications such as terminal emulators and protocols such as PPP can run unchanged over the virtual serial interfaces created by RFCOMM.

Device Example: Pill Dispenser

To take an example, assume that you have a Bluetooth-aware pill dispenser. When you pop a pill out of the dispenser, it sends a message over a Bluetooth RFCOMM channel. A Linux cell phone, such as the one shown in Figure 6.5 of Chapter 6 , reads this alert using a simple application that establishes an RFCOMM connection to the pill dispenser. The phone then dispatches this information to the health-care provider's server on the Internet via its GPRS interface.

A skeletal application on the Linux cell phone that reads data arriving from the pill dispenser using the BlueZ socket API is shown in Listing 16.1 . The listing assumes that you are familiar with the basics of socket programming.

Listing 16.1. Communicating with a Pill Dispenser over RFCOMM

Code View:

```
#include <sys/socket.h>
#include <bluetooth/rfcomm.h>  /* For struct sockaddr_rc */

void
sense_dispenser()
{
  int pillfd;
  struct sockaddr_rc pill_rfcomm;
  char buffer[1024];

  /* ... */

  /* Create a Bluetooth RFCOMM socket */
  if ((pillfd = socket(PF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM))
       < 0) {
```

```
    printf("Bad Bluetooth RFCOMM socket");
    exit(1);
  }

  /* Connect to the pill dispenser */
  pill_rfcomm.rc_family  = AF_BLUETOOTH;
  pill_rfcomm.rc_bdaddr  = PILL_DISPENSER_BLUETOOTH_ADDR;
  pill_rfcomm.rc_channel = PILL_DISPENSER_RFCOMM_CHANNEL;

  if (connect(pillfd, (struct sockaddr *)&pill_rfcomm,
              sizeof(pill_rfcomm))) {
    printf("Cannot connect to Pill Dispenser\n");
    exit(1);
  }

  printf("Connection established to Pill Dispenser\n");
  /* Poll until data is ready */
  select(pillfd, &fds, NULL, NULL, &timeout);

  /* Data is available on this RFCOMM channel */
  if (FD_ISSET(pillfd, fds)) {

    /* Read pill removal alerts from the dispenser */
    read(pillfd, buffer, sizeof(buffer));

    /* Take suitable action; e.g., send a message to the health
       care provider's server on the Internet via the GPRS
       interface */
    /* ... */
  }

  /* ... */
}
```

Networking

Trace down the code path from the *telnet/ftp/ssh* box in Figure 16.2 to see how networking is architected over BlueZ Bluetooth. As you can see, there are two different ways to network over Bluetooth:

1. By running TCP/IP directly over BNEP. The resulting network is called a *personal area network* (PAN).

2. By running TCP/IP over PPP over RFCOMM. This is called *dialup networking* (DUN).

The kernel implementation of Bluetooth networking is unlikely to interest the device driver writer and is not explored. Table 16.2 shows the steps required to network two laptops using PAN, however. Networking with DUN resembles this and is not examined. The laptops are respectively Bluetooth-enabled using the Sharp CF card and the Belkin USB adapter discussed earlier. You can slip the CF card into the first laptop's PCMCIA slot using a passive CF-to-PCMCIA adapter. Look at Figure 16.2 in tandem with Table 16.2 to understand the mappings to corresponding BlueZ components. Table 16.2 uses **bash-sharp>** and **bash-belkin>** as the respective shell prompts of the two laptops.

On the laptop with the Sharp Bluetooth CF card

1. Start the HCI and service discovery daemons:

   ```
   bash-sharp> hcid
   bash-sharp> sdpd
   ```

   Because this device possesses a UART interface, you have to attach the BlueZ stack to the appropriate serial port. In this case, assume that *serial_cs* has allotted */dev/ttyS3* to the card:

   ```
   bash-sharp> hciattach ttyS3 any
   ```

2. Verify that the HCI interface is up:

   ```
   bash-sharp> hciconfig -a
    hci0:   Type: UART
    BD Address: 08:00:1F:10:3B:13 ACL MTU: 60:20  SCO MTU: 31:1
    UP RUNNING PSCAN ISCAN
    ...
   Manufacturer: Silicon Wave (11)
   ```

3. Verify that basic BlueZ modules are loaded:

   ```
   bash-sharp> lsmod
    Module                  Size  Used by
    hci_uart               16728  3
    l2cap                  26144  2
    bluetooth              47684  6 hci_uart,l2cap
    ...
   ```

4. Insert the BlueZ module that implements network encapsulation:

   ```
   bash-sharp> modprobe bnep
   ```

5. Listen for incoming PAN connections:[1]

   ```
   bash-sharp> pand –s
   ```

On the laptop with the Belkin USB Bluetooth adapter

1. Start daemons, such as hcid and sdpd, and insert necessary kernel modules, such as *bluetooth.ko* and *l2cap.ko* .

2. Because this is a USB device, you don't need to invoke hciattach, but make sure that the *hci_usb.ko* module is inserted.

3. Verify that the HCI interface is up:

   Code View:
   ```
   bash-belkin> hciconfig -a
    hci0:   Type: USB BD Address: 00:02:72:B0:33:AB ACL MTU: 192:8  SCO MTU: 64:8
    UP RUNNING PSCAN ISCAN
   ```

```
...
Manufacturer: Cambridge Silicon Radio (10)
```

4. Search and discover devices in the neighborhood:

```
bash-belkin> hcitool -i hci0 scan --flush
 Scanning....
 08:00:1F:10:3B:13 bash-sharp
```

5. Establish a PAN with the first laptop. You can get its Bluetooth address (08:00:1F:10:3B:13) from its `hciconfig` output:

```
bash-belkin> pand -c 08:00:1F:10:3B:13
```

If you now look at the `ifconfig` output on the two laptops, you will find that a new interface named `bnep0` has made an appearance at both ends. Assign IP addresses to both interfaces and get ready to telnet and FTP!

Table 16.2. Networking Two Laptops Using Bluetooth PAN

[1] A useful command-line option to `pand` is `--persist` , which automatically attempts to reconnect when a connection drops. Dig into the man pages for more invocation options.

Human Interface Devices

Look at sections "USB and Bluetooth Keyboards " and "USB and Bluetooth Mice " in Chapter 7 , "Input Drivers," for a discussion on Bluetooth human interface devices.

Audio

Let's take the example of an HBH-30 Sony Ericsson Bluetooth headset to understand Bluetooth SCO audio. Before the headset can start communicating with a Linux device, the Bluetooth link layer on the latter has to discover the former. For this, put the headset in *discover* mode by pressing the button earmarked for device discovery. In addition, you have to configure BlueZ with the headset's *personal identification number* (PIN) by adding it to */etc/bluetooth/pin.* An application on the Linux device that uses BlueZ SCO APIs can now send audio data to the headset. The audio data should be in a format that the headset understands. The HBH-30 uses the A-law PCM (*pulse code modulation*) format. There are public domain utilities for converting audio into various PCM formats.

Bluetooth chipsets commonly have PCM interface pins in addition to the HCI transport interface. If a device supports, for instance, both Bluetooth and *Global System for Mobile Communication* (GSM), the PCM lines from the GSM chipset may be directly wired to the Bluetooth chip's PCM audio lines. You might then have to configure the Bluetooth chip to receive and send SCO audio packets over its HCI interface instead of its PCM interface.

Debugging

There are two BlueZ tools useful for debugging:

1. *hcidump* taps HCI packets flowing back and forth, and parses them into human-readable form. Here's an example dump while a device inquiry is in progress:

```
bash> hcidump -i hci0
HCIDump - HCI packet analyzer ver 1.11
device: hci0 snap_len: 1028 filter: 0xffffffff
   HCI Command: Inquiry (0x01|0x0001) plen 5
   HCI Event: Command Status (0x0f) plen 4
   HCI Event: Inquiry Result (0x02) plen 15
   ...
   HCI Event: Inquiry Complete (0x01) plen 1 < HCI Command:
   Remote Name Request (0x01|0x0019) plen 10
   ...
```

2. The virtual HCI driver (*hci_vhci.ko*), as shown in Figure 16.2 , emulates a Bluetooth interface if you do not have actual hardware.

Looking at the Sources

Look inside *drivers/bluetooth/* for BlueZ low-level drivers. Explore *net/bluetooth/* for insights into the BlueZ protocol implementation.

Bluetooth applications fall under different *profiles* based on how they behave. For example, the cordless telephony profile specifies how a Bluetooth device can implement a cordless phone. We discussed profiles for PAN and serial access, but there are many more profiles out there such as fax profile, *General Object Exchange Profile* (GOEP) and *SIM Access Profile* (SAP). The *bluez-utils* package, downloadable from www.bluez.org , provides support for several Bluetooth profiles.

The official Bluetooth website is www.bluetooth.org . It contains Bluetooth specification documents and information about the *Bluetooth Special Interest Group* (SIG).

Affix is an alternate Bluetooth stack on Linux. You can download Affix from http://affix.sourceforge.net/ .

Chapter 16. Linux Without Wires

In This Chapter

- Bluetooth

467

- Infrared

478

- WiFi

489

- Cellular Networking

496

- Current Trends

500

Several small-footprint devices are powered by the dual combination of a wireless technology and Linux. Bluetooth, Infrared, WiFi, and cellular networking are established wireless technologies that have healthy Linux support. Bluetooth eliminates cables, injects intelligence into dumb devices, and opens a flood gate of novel applications. Infrared is a low-cost, low-range, medium-rate, wireless technology that can network laptops, connect handhelds, or dispatch a document to a printer. WiFi is the wireless equivalent of an Ethernet LAN. Cellular networking using *General Packet Radio Service* (GPRS) or *code division multiple access* (CDMA) keeps you Internet-enabled on the go, as long as your wanderings are confined to service provider coverage area.

Because these wireless technologies are widely available in popular form factors, you are likely to end up, sooner rather than later, with a card that does not work on Linux right away. Before you start working on enabling an unsupported card, you need to know in detail how the kernel implements support for the corresponding technology. In this chapter, let's learn how Linux enables Bluetooth, Infrared, WiFi, and cellular networking.

Wireless Trade-Offs

Bluetooth, Infrared, WiFi, and GPRS serve different niches. The trade-offs can be gauged in terms of speed, range, cost, power consumption, ease of hardware/software co-design, and PCB real estate usage.

Table 16.1 gives you an idea of these parameters, but you will have to contend with several variables when you measure the numbers on the ground. The speeds listed are the theoretical maximums. The power consumptions indicated are relative, but in the real world they also depend on the vendor's implementation techniques, the technology subclass, and the operating mode. Cost economics depend on the chip form factor and whether the chip contains built-in microcode that implements some of the protocol layers. The board real estate consumed depends not just on the chipset, but also on transceivers, antennae, and whether you build using *off-the-shelf* (OTS) modules.

Bluetooth
720Kbps
10m to 100m
**
**
**
**
Infrared Data
4Mbps (Fast IR)
Up to 1 meter within a 30-degree cone
*
*
*
*
WiFi
54Mbps
150 meters (indoors)
****
***
***
***
GPRS
170Kbps
Service provider coverage
***
****
*
***

Note: The last four columns give relative measurement (depending on the number of * symbols) rather than absolute values.

Table 16.1. Wireless Trade-Offs

	Speed	Range	Power	Cost	Co-Design Effort	Board Real Estate

Some sections in this chapter focus more on "system programming" than device drivers. This is because the corresponding regions of the protocol stack (for example, Bluetooth RFCOMM and Infrared networking) are already present in the kernel and you are more likely to perform associated user mode customizations than develop protocol content or device drivers.

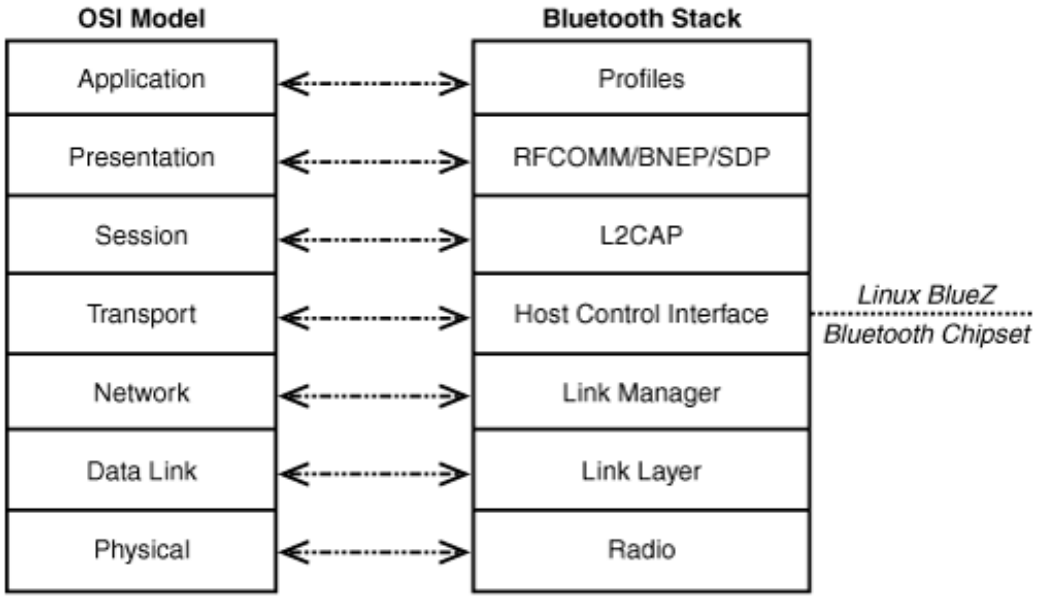
Bluetooth

Bluetooth is a short-range cable-replacement technology that carries both data and voice. It supports speeds of up to 723Kbps (asymmetric) and 432Kbps (symmetric). *Class 3* Bluetooth devices have a range of 10 meters, and *Class 1* transmitters can communicate up to 100 meters.

Bluetooth is designed to do away with wires that constrict and clutter your environment. It can, for example, turn your wristwatch into a front-end for a bulky *Global Positioning System* (GPS) hidden inside your backpack. Or it can, for instance, let you navigate a presentation via your handheld. Again, Bluetooth can be the answer if you want your laptop to be a hub that can Internet-enable your Bluetooth-aware MP3 player. If your wristwatch, handheld, laptop, or MP3 player is running Linux, knowledge of the innards of the Linux Bluetooth stack will help you extract maximum mileage out of your device.

As per the Bluetooth specification, the protocol stack consists of the layers shown in Figure 16.1 . The radio, link controller, and link manager roughly correspond to the physical, data link, and network layers in the *Open Systems Interconnect* (OSI) standard reference model. The *Host Control Interface* (HCI) is the protocol that carries data to/from the hardware and, hence, maps to the transport layer. The Bluetooth *Logical Link Control and Adaptation Protocol* (L2CAP) falls in the session layer. Serial port emulation using *Radio Frequency Communication* (RFCOMM), Ethernet emulation using *Bluetooth Network Encapsulation Protocol* (BNEP), and the *Service Discovery Protocol* (SDP) are part of the feature-rich presentation layer. At the top of the stack reside various application environments called *profiles.* The radio, link controller, and link manager are usually part of Bluetooth hardware, so operating system support starts at the HCI layer.

Figure 16.1. The Bluetooth stack.



A common method of interfacing Bluetooth hardware with a microcontroller is by connecting the chipset's data lines to the controller's UART pins. Figure 13.4 of Chapter 13 , "Audio Drivers," shows a Bluetooth chip on an MP3 player communicating with the processor via a UART. USB is another oft-used vehicle for communicating with Bluetooth chipsets. Figure 11.2 of Chapter 11 , "Universal Serial Bus," shows a Bluetooth chip on an embedded device interfacing with the processor over USB. Irrespective of whether you use UART or USB (we will look at both kinds of devices later), the packet format used to transport Bluetooth data is HCI.

BlueZ

The BlueZ Bluetooth implementation is part of the mainline kernel and is the official Linux Bluetooth stack.

Figure 16.2 shows how BlueZ maps Bluetooth protocol layers to kernel modules, kernel threads, user space daemons, configuration tools, utilities, and libraries. The main BlueZ components are explained here:

1. *bluetooth.ko* contains the core BlueZ infrastructure. All other BlueZ modules utilize its services. It's also responsible for exporting the Bluetooth family of sockets (`AF_BLUETOOTH`) to user space and for populating related sysfs entries.

2. For transporting Bluetooth HCI packets over UART, the corresponding BlueZ HCI implementation is *hci_uart.ko.* For USB transport, it's *hci_usb.ko* .

3. *l2cap.ko* implements the L2CAP adaptation layer that is responsible for segmentation and reassembly. It also multiplexes between different higher-layer protocols.

4. To run TCP/IP applications over Bluetooth, you have to emulate Ethernet ports over L2CAP using BNEP. This is accomplished by *bnep.ko.* To service BNEP connections, BlueZ spawns a kernel thread called *kbnepd* .

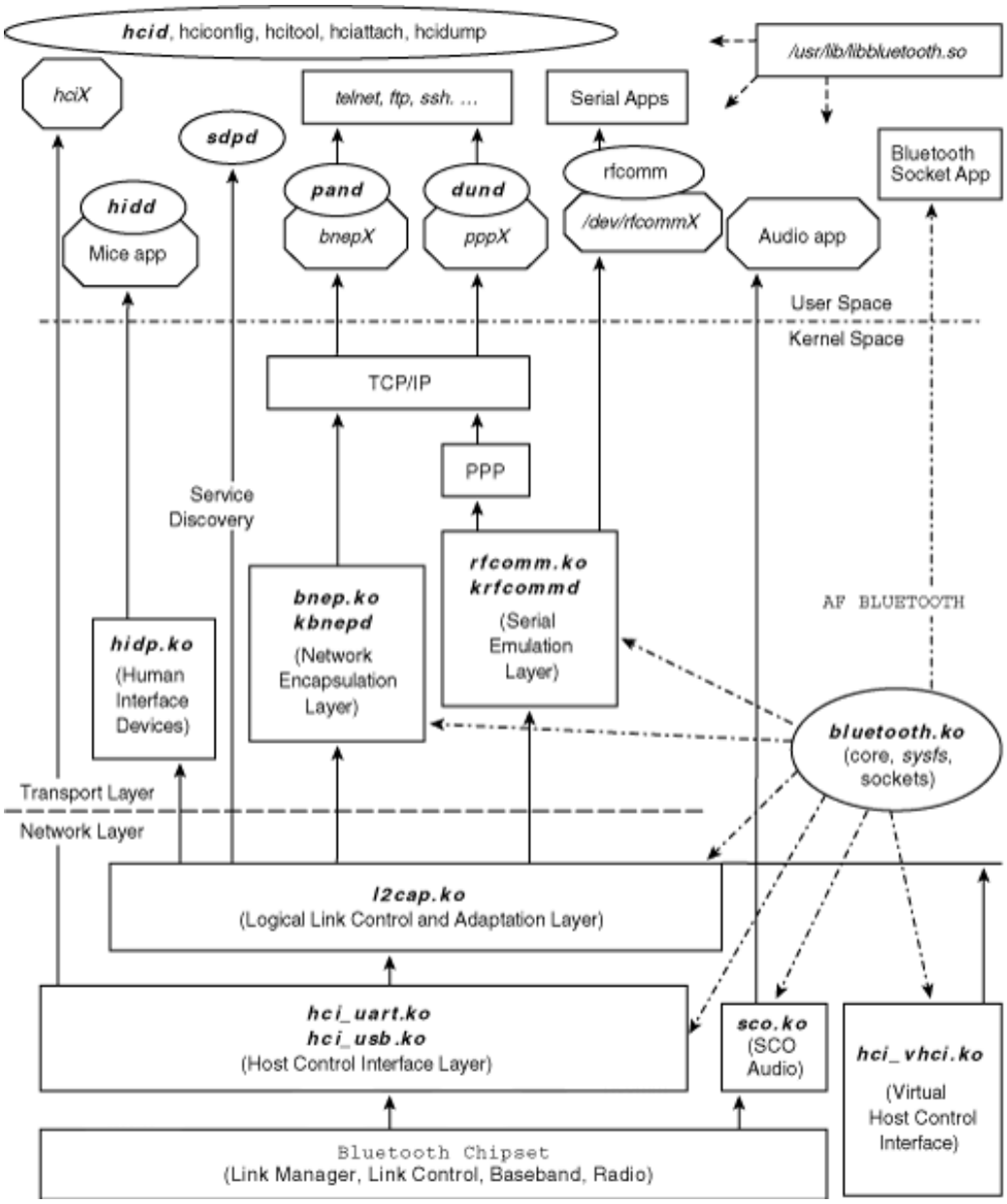
5. To run serial port applications such as terminal emulators over Bluetooth, you need to emulate serial ports over L2CAP. This is accomplished by *rfcomm.ko.* RFCOMM also functions as the pillar that supports networking over PPP. To service incoming RFCOMM connections, *rfcomm.ko* spawns a kernel thread called *krfcommd.* To set up and maintain connections to individual RFCOMM channels on target devices, use the *rfcomm* utility.

6. The *Human Interface Devices* (HID) layer is implemented via *hidp.ko* . The user mode daemon, *hidd* , lets BlueZ handle input devices such as Bluetooth mice.

7. Audio is handled via the *Synchronous Connection Oriented* (SCO) layer implemented by *sco.ko* .

Figure 16.2. Bluetooth protocol layers mapped to BlueZ kernel modules.

[View full size image]



Let's now trace the kernel code flow for two example Bluetooth devices: a *Compact Flash* (CF) card and a USB adapter.

Device Example: CF Card

The Sharp Bluetooth Compact Flash card is built using a Silicon Wave chipset and uses a serial transport to carry HCI packets. There are three different ways by which HCI packets can be transported serially:

1. H4 (UART), which is used by the Sharp CF card. H4 is the standard method to transfer Bluetooth data over UARTs as defined by the Bluetooth specification. Look at *drivers/bluetooth/hci_h4.c* for the BlueZ implementation.

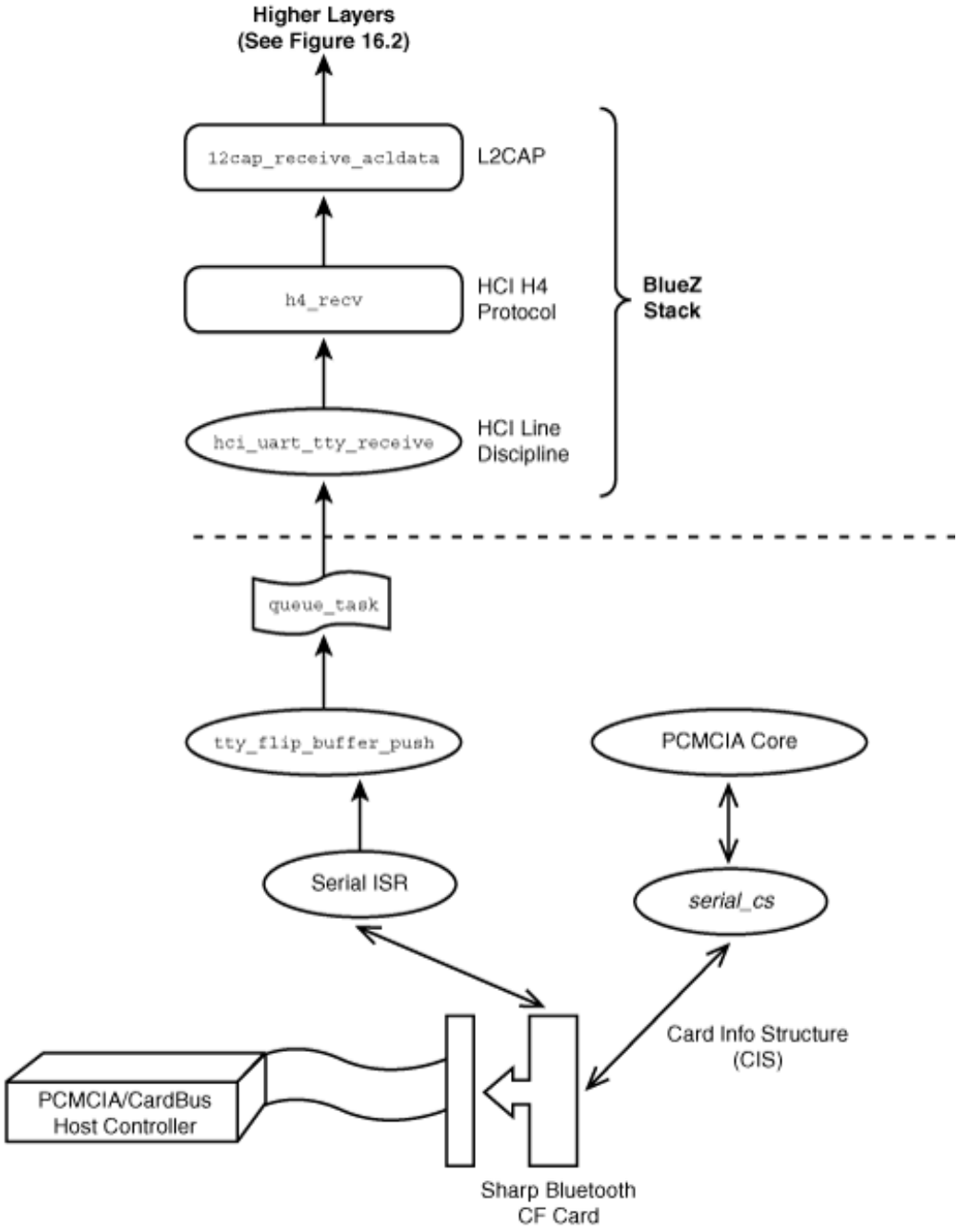
2. H3 (RS232). Devices using H3 are hard to find. BlueZ has no support for H3.

3. *BlueCore Serial Protocol* (BCSP), which is a proprietary protocol from *Cambridge Silicon Radio* (CSR) that supports error checking and retransmission. BCSP is used on non-USB devices based on CSR BlueCore chips including PCMCIA and CF cards. The BlueZ BCSP implementation lives in *drivers/-bluetooth/hci_bcsp.c* .

The read data path for the Sharp Bluetooth card is shown in Figure 16.3 . The first point of contact between the card and the kernel is at the UART driver. As you saw in Figure 9.5 of Chapter 9 , "PCMCIA and Compact Flash," the serial Card Services driver, *drivers/serial/serial_cs.c* , allows the rest of the operating system to see the Sharp card as if it were a serial device. The serial driver passes on the received HCI packets to BlueZ. BlueZ implements HCI processing in the form of a kernel line discipline. As you learned in Chapter 6 , "Serial Drivers," line disciplines reside above the serial driver and shape its behavior. The HCI line discipline invokes associated protocol routines (H4 in this case) for assistance in data processing. From then on, L2CAP and higher BlueZ layers take charge.

Figure 16.3. Read data path from a Sharp Bluetooth CF card.

[View full size image]



Device Example: USB Adapter

Let's now look at a device that uses USB to transport HCI packets. The Belkin Bluetooth USB adapter is one such gadget. In this case, the Linux USB layer (*drivers/usb/**), the HCI USB transport driver (*drivers/bluetooth/hci_usb.c*), and the BlueZ protocol stack (*net/bluetooth/**) are the main players that get the data rolling. Let's see how these three kernel layers interact.

As you learned in Chapter 11 , USB devices exchange data using one or more of four pipes. For Bluetooth USB devices, each pipe is responsible for carrying a particular type of data:

1. Control pipes are used to transport HCI commands.

2. Interrupt pipes are responsible for carrying HCI events.

3. Bulk pipes transfer *asynchronous connectionless* (ACL) Bluetooth data.

4. Isochronous pipes carry SCO audio data.

You also saw in Chapter 11 that when a USB device is plugged into a system, the host controller driver enumerates it using a control pipe and assigns endpoint addresses between 1 and 127. The configuration descriptor read by the USB subsystem during enumeration contains information about the device, such as its `class` , `subclass` , and `protocol` . The Bluetooth specification defines the (`class` , `subclass` , `protocol`) codes of Bluetooth USB devices as (`0xE, 0x01, 0x01`) . The HCI USB transport driver (*hci_usb*) registers these values with the USB core during initialization. When the Belkin USB adapter is plugged in, the USB core reads the (`class` , `subclass` , `protocol`) information from the device configuration descriptor. Because this information matches the values registered by *hci_usb* , this driver gets attached to the Belkin USB adapter. *hci_usb* reads Bluetooth data from the four USB pipes described previously and passes it on to the BlueZ protocol stack. Linux applications now run seamlessly over this device, as shown in Figure 16.2 .

RFCOMM

RFCOMM emulates serial ports over Bluetooth. Applications such as terminal emulators and protocols such as PPP can run unchanged over the virtual serial interfaces created by RFCOMM.

Device Example: Pill Dispenser

To take an example, assume that you have a Bluetooth-aware pill dispenser. When you pop a pill out of the dispenser, it sends a message over a Bluetooth RFCOMM channel. A Linux cell phone, such as the one shown in Figure 6.5 of Chapter 6 , reads this alert using a simple application that establishes an RFCOMM connection to the pill dispenser. The phone then dispatches this information to the health-care provider's server on the Internet via its GPRS interface.

A skeletal application on the Linux cell phone that reads data arriving from the pill dispenser using the BlueZ socket API is shown in Listing 16.1 . The listing assumes that you are familiar with the basics of socket programming.

Listing 16.1. Communicating with a Pill Dispenser over RFCOMM

Code View:

```
#include <sys/socket.h>
#include <bluetooth/rfcomm.h>  /* For struct sockaddr_rc */

void
sense_dispenser()
{
  int pillfd;
  struct sockaddr_rc pill_rfcomm;
  char buffer[1024];

  /* ... */

  /* Create a Bluetooth RFCOMM socket */
  if ((pillfd = socket(PF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM))
       < 0) {
```

```
    printf("Bad Bluetooth RFCOMM socket");
    exit(1);
  }

  /* Connect to the pill dispenser */
  pill_rfcomm.rc_family  = AF_BLUETOOTH;
  pill_rfcomm.rc_bdaddr  = PILL_DISPENSER_BLUETOOTH_ADDR;
  pill_rfcomm.rc_channel = PILL_DISPENSER_RFCOMM_CHANNEL;

  if (connect(pillfd, (struct sockaddr *)&pill_rfcomm,
              sizeof(pill_rfcomm))) {
    printf("Cannot connect to Pill Dispenser\n");
    exit(1);
  }

  printf("Connection established to Pill Dispenser\n");
  /* Poll until data is ready */
  select(pillfd, &fds, NULL, NULL, &timeout);

  /* Data is available on this RFCOMM channel */
  if (FD_ISSET(pillfd, fds)) {

    /* Read pill removal alerts from the dispenser */
    read(pillfd, buffer, sizeof(buffer));

    /* Take suitable action; e.g., send a message to the health
       care provider's server on the Internet via the GPRS
       interface */
    /* ... */
  }

  /* ... */
}
```

Networking

Trace down the code path from the *telnet/ftp/ssh* box in Figure 16.2 to see how networking is architected over BlueZ Bluetooth. As you can see, there are two different ways to network over Bluetooth:

1. By running TCP/IP directly over BNEP. The resulting network is called a *personal area network* (PAN).

2. By running TCP/IP over PPP over RFCOMM. This is called *dialup networking* (DUN).

The kernel implementation of Bluetooth networking is unlikely to interest the device driver writer and is not explored. Table 16.2 shows the steps required to network two laptops using PAN, however. Networking with DUN resembles this and is not examined. The laptops are respectively Bluetooth-enabled using the Sharp CF card and the Belkin USB adapter discussed earlier. You can slip the CF card into the first laptop's PCMCIA slot using a passive CF-to-PCMCIA adapter. Look at Figure 16.2 in tandem with Table 16.2 to understand the mappings to corresponding BlueZ components. Table 16.2 uses **bash-sharp>** and **bash-belkin>** as the respective shell prompts of the two laptops.

On the laptop with the Sharp Bluetooth CF card

1. Start the HCI and service discovery daemons:

   ```
   bash-sharp> hcid
   bash-sharp> sdpd
   ```

   Because this device possesses a UART interface, you have to attach the BlueZ stack to the appropriate serial port. In this case, assume that *serial_cs* has allotted */dev/ttyS3* to the card:

   ```
   bash-sharp> hciattach ttyS3 any
   ```

2. Verify that the HCI interface is up:

   ```
   bash-sharp> hciconfig -a
    hci0:   Type: UART
    BD Address: 08:00:1F:10:3B:13 ACL MTU: 60:20  SCO MTU: 31:1
    UP RUNNING PSCAN ISCAN
    ...
   Manufacturer: Silicon Wave (11)
   ```

3. Verify that basic BlueZ modules are loaded:

   ```
   bash-sharp> lsmod
    Module                  Size  Used by
    hci_uart               16728  3
    l2cap                  26144  2
    bluetooth              47684  6 hci_uart,l2cap
    ...
   ```

4. Insert the BlueZ module that implements network encapsulation:

   ```
   bash-sharp> modprobe bnep
   ```

5. Listen for incoming PAN connections:[1]

   ```
   bash-sharp> pand –s
   ```

On the laptop with the Belkin USB Bluetooth adapter

1. Start daemons, such as hcid and sdpd, and insert necessary kernel modules, such as *bluetooth.ko* and *l2cap.ko* .

2. Because this is a USB device, you don't need to invoke hciattach, but make sure that the *hci_usb.ko* module is inserted.

3. Verify that the HCI interface is up:

   Code View:
   ```
   bash-belkin> hciconfig -a
    hci0:   Type: USB BD Address: 00:02:72:B0:33:AB ACL MTU: 192:8  SCO MTU: 64:8
    UP RUNNING PSCAN ISCAN
   ```

```
...
Manufacturer: Cambridge Silicon Radio (10)
```

4. Search and discover devices in the neighborhood:

```
bash-belkin> hcitool -i hci0 scan --flush
 Scanning....
 08:00:1F:10:3B:13 bash-sharp
```

5. Establish a PAN with the first laptop. You can get its Bluetooth address (08:00:1F:10:3B:13) from its `hciconfig` output:

```
bash-belkin> pand -c 08:00:1F:10:3B:13
```

If you now look at the `ifconfig` output on the two laptops, you will find that a new interface named `bnep0` has made an appearance at both ends. Assign IP addresses to both interfaces and get ready to telnet and FTP!

Table 16.2. Networking Two Laptops Using Bluetooth PAN

[1] A useful command-line option to `pand` is `--persist` , which automatically attempts to reconnect when a connection drops. Dig into the man pages for more invocation options.

Human Interface Devices

Look at sections "USB and Bluetooth Keyboards " and "USB and Bluetooth Mice " in Chapter 7 , "Input Drivers," for a discussion on Bluetooth human interface devices.

Audio

Let's take the example of an HBH-30 Sony Ericsson Bluetooth headset to understand Bluetooth SCO audio. Before the headset can start communicating with a Linux device, the Bluetooth link layer on the latter has to discover the former. For this, put the headset in *discover* mode by pressing the button earmarked for device discovery. In addition, you have to configure BlueZ with the headset's *personal identification number* (PIN) by adding it to */etc/bluetooth/pin.* An application on the Linux device that uses BlueZ SCO APIs can now send audio data to the headset. The audio data should be in a format that the headset understands. The HBH-30 uses the A-law PCM (*pulse code modulation*) format. There are public domain utilities for converting audio into various PCM formats.

Bluetooth chipsets commonly have PCM interface pins in addition to the HCI transport interface. If a device supports, for instance, both Bluetooth and *Global System for Mobile Communication* (GSM), the PCM lines from the GSM chipset may be directly wired to the Bluetooth chip's PCM audio lines. You might then have to configure the Bluetooth chip to receive and send SCO audio packets over its HCI interface instead of its PCM interface.

Debugging

There are two BlueZ tools useful for debugging:

1. *hcidump* taps HCI packets flowing back and forth, and parses them into human-readable form. Here's an example dump while a device inquiry is in progress:

```
bash> hcidump -i hci0
HCIDump - HCI packet analyzer ver 1.11
device: hci0 snap_len: 1028 filter: 0xffffffff
   HCI Command: Inquiry (0x01|0x0001) plen 5
   HCI Event: Command Status (0x0f) plen 4
   HCI Event: Inquiry Result (0x02) plen 15
   ...
   HCI Event: Inquiry Complete (0x01) plen 1 < HCI Command:
   Remote Name Request (0x01|0x0019) plen 10
   ...
```

2. The virtual HCI driver (*hci_vhci.ko*), as shown in Figure 16.2 , emulates a Bluetooth interface if you do not have actual hardware.

Looking at the Sources

Look inside *drivers/bluetooth/* for BlueZ low-level drivers. Explore *net/bluetooth/* for insights into the BlueZ protocol implementation.

Bluetooth applications fall under different *profiles* based on how they behave. For example, the cordless telephony profile specifies how a Bluetooth device can implement a cordless phone. We discussed profiles for PAN and serial access, but there are many more profiles out there such as fax profile, *General Object Exchange Profile* (GOEP) and *SIM Access Profile* (SAP). The *bluez-utils* package, downloadable from www.bluez.org , provides support for several Bluetooth profiles.

The official Bluetooth website is www.bluetooth.org . It contains Bluetooth specification documents and information about the *Bluetooth Special Interest Group* (SIG).

Affix is an alternate Bluetooth stack on Linux. You can download Affix from http://affix.sourceforge.net/ .

Infrared

Infrared (IR) rays are optical waves lying between the visible and the microwave regions of the electromagnetic spectrum. One use of IR is in point-to-point data communication. Using IR, you can exchange visiting cards between PDAs, network two laptops, or dispatch a document to a printer. IR has a range of up to 1 meter within a 30-degree cone, spreading from –15 to +15 degrees.

There are two popular flavors of IR communication: *Standard IR* (SIR), which supports speeds of up to 115.20 Kbaud; and *Fast IR* (FIR), which has a bandwidth of 4Mbps.

Figure 16.4 shows IR connection on a laptop. UART1 in the Super I/O chipset is IR-enabled, so an IR transceiver is directly connected to it. Laptops having no IR support in their Super I/O chip may rely on an external IR dongle (see the section "Device Example: IR Dongle") similar to the one connected to UART0. Figure 16.5 shows IR connection on an embedded SoC having a built-in IR dongle connected to a system UART.

Figure 16.4. IrDA on a laptop.

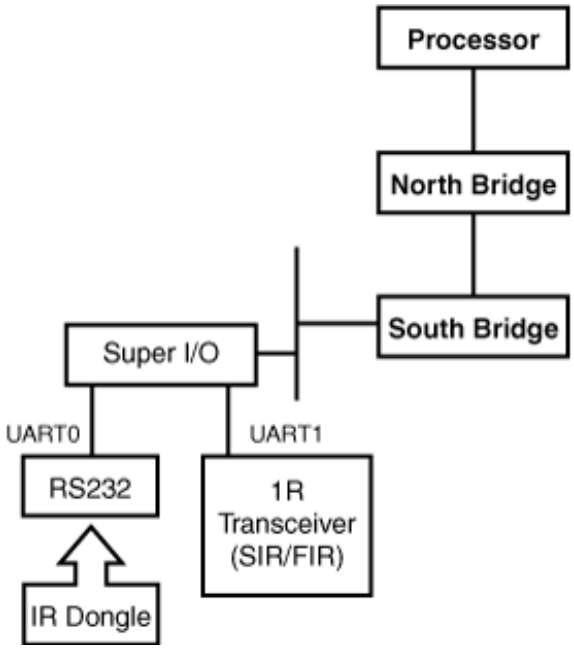
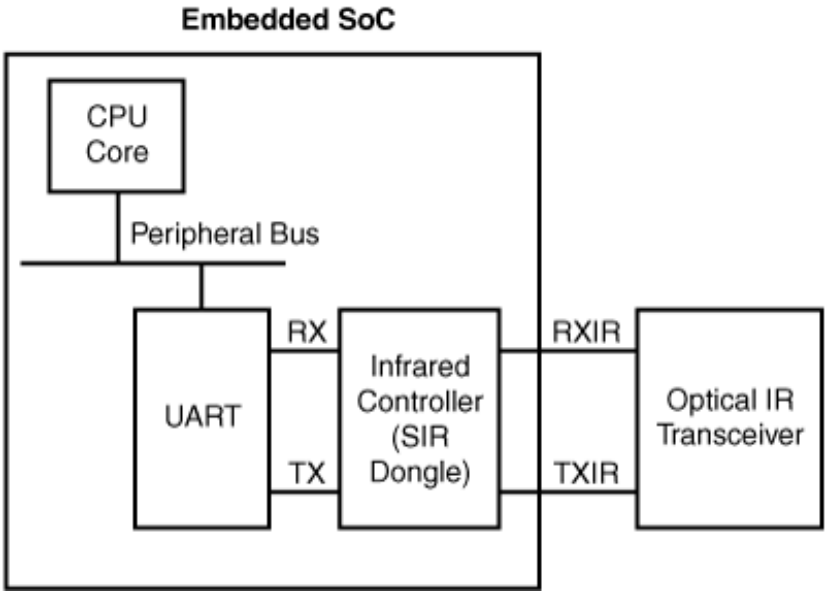


Figure 16.5. IrDA on an embedded device (for example, EP7211).



Linux supports IR communication on two planes:

1. Intelligent data-transfer via protocols specified by the *Infrared Data Association* (IrDA). This is implemented by the Linux-IrDA project.

2. Controlling applications via a remote control. This is implemented by the *Linux Infrared Remote Control* (LIRC) project.

This section primarily explores Linux-IrDA but takes a quick look at LIRC before wrapping up.

Linux-IrDA

The Linux-IrDA project (http://irda.sourceforge.net/) brings IrDA capabilities to the kernel. To get an idea of how Linux-IrDA components relate vis-à-vis the IrDA stack and possible hardware configurations, let's criss-cross through Figure 16.6:

1. Device drivers constitute the bottom layer. SIR chipsets that are 16550-compatible can reuse the native Linux serial driver after shaping its behavior using the IrDA line discipline, IrTTY. An alternative to this combo is the IrPort driver. FIR chipsets have their own special drivers.

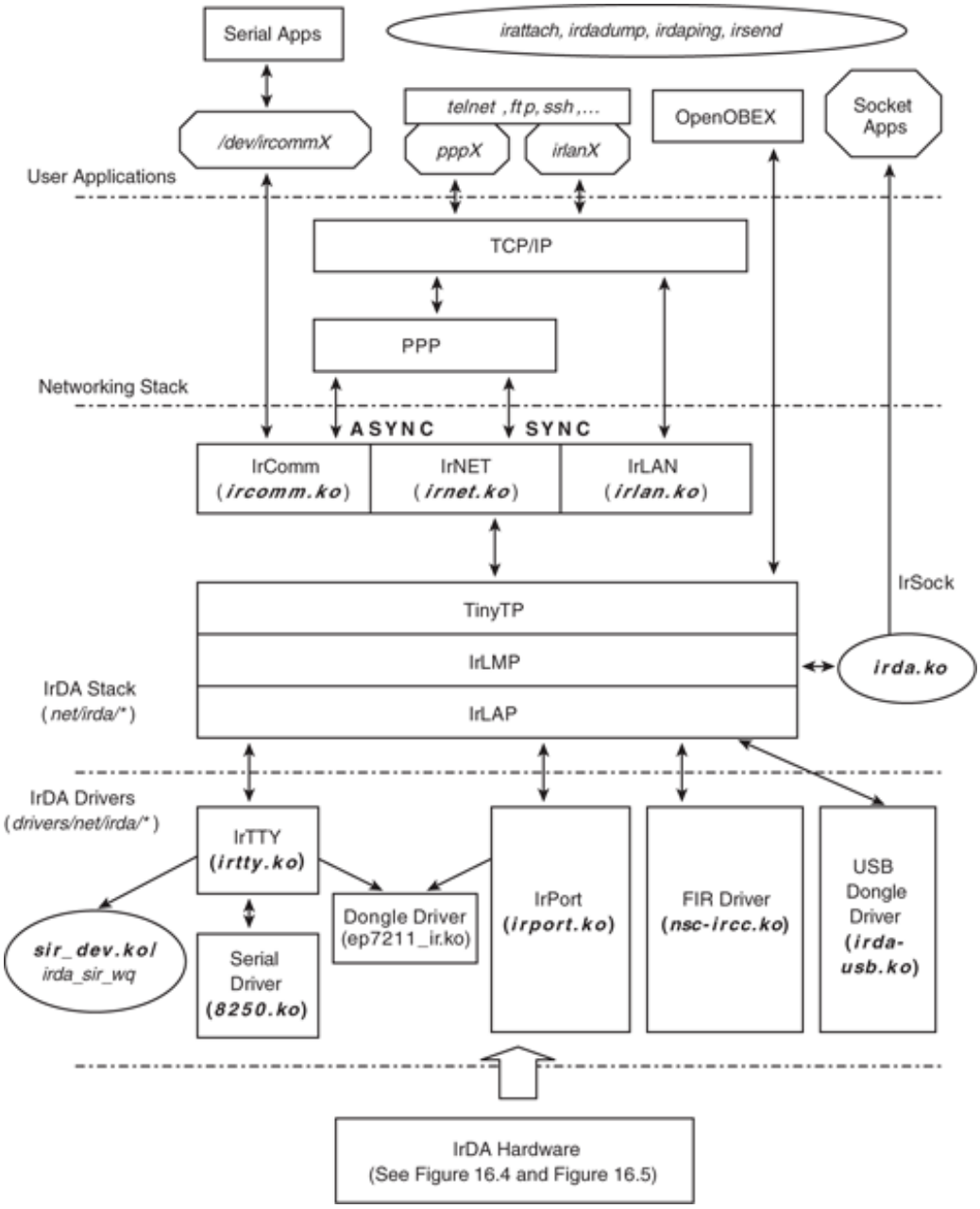
2. Next comes the core protocol stack. This consists of the *IR Link Access Protocol* (IrLAP), *IR Link Management Protocol* (IrLMP), *Tiny Transport Protocol* (TinyTP), and the *IrDA socket* (IrSock) interface. IrLAP provides a reliable transport as well as the state machine to discover neighboring devices. IrLMP is a multiplexer over IrLAP. TinyTP provides segmentation, reassembly, and flow control. IrSock offers a socket interface over IrLMP and TinyTP.

3. Higher regions of the stack marry IrDA to data-transfer applications. IrLAN and IrNET enable networking, while IrComm allows serial communication.

4. You also need the applications that ultimately make or break the technology. An example is *openobex* (http://openobex.sourceforge.net/), which implements the *OBject EXchange* (OBEX) protocol used to exchange objects such as documents and visiting cards. To configure Linux-IrDA, you need the *irda-utils* package that comes bundled with many distributions. This provides tools such as *irattach*, *irdadump*, and *irdaping*.

Figure 16.6. Communicating over Linux-IrDA.

[View full size image]



Device Example: Super I/O Chip

To get a first taste of Linux-IrDA, let's get two laptops talking to each other over IR. Each laptop is IR-enabled via National Semiconductor's NSC PC87382 Super I/O chip.[2] UART1 in Figure 16.4 shows the connection scenario. The PC87382 chip can work in both SIR and FIR modes. We will look at each in turn.

[2] Super I/O chipsets typically support several peripherals besides IrDA, such as serial ports, parallel ports, *Musical Instrument Digital Interface* (MIDI), and floppy controllers.

SIR chips offer a UART interface to the host computer. For communicating in SIR mode, attach the associated UART port (*/dev/ttyS1* in this example) of each laptop to the IrDA stack:

```
bash> irattach /dev/ttyS1 -s
```

Verify that IrDA kernel modules (*irda.ko*, *sir_dev.ko*, and *irtty_sir.ko*) are loaded and that the *irda_sir_wq* kernel thread is running. The irda0 interface should also have made an appearance in the `ifconfig` output. The `-s` option to `irattach` triggers a search for IR activity in the neighborhood. If you slide the laptops such that their IR transceivers lie within the range cone, they will be able to spot each other:

```
bash> cat /proc/net/irda/discovery
nickname: localhost, hint: 0x4400, saddr: 0x55529048, daddr: 0x8fefb350
```

The other laptop makes a similar announcement, but with the source and destination addresses (`saddr` and `daddr`) reversed. You may set the desired communication speed using stty on ttyS1. To set the baud rate to 19200, do this:

```
bash> stty speed 19200 < /dev/ttyS1
```

The easiest way to cull IR activity from the air is by using the debug tool, *irdadump*. Here's a sample dump obtained during the preceding connection establishment, which shows the negotiated parameters:

Code View:
```
bash> irdadump -i irda0
...
22:05:07.831424 snrm:cmd ca=fe pf=1 6fb7ff33 > 2c0ce8b6 new-ca=40
LAP QoS: Baud Rate=19200bps Max Turn Time=500ms Data Size=2048B Window Size=7 Add
BOFS=0 Min Turn Time=5000us Link Disc=12s (32)
22:05:07.987043 ua:rsp ca=40 pf=1 6fb7ff33 < 2c0ce8b6
LAP QoS: Baud Rate=19200bps Max Turn Time=500ms Data Size=2048B Window Size=7 Add
BOFS=0 Min Turn Time=5000us Link Disc=12s (31)
...
```

You can also obtain debug information out of the IrDA stack by controlling the verbosity level in */proc/sys/net/irda/debug*.

To set the laptops in FIR mode, dissociate ttyS1 from the native serial driver and instead attach it to the NSC FIR driver, *nsc-ircc.ko*:

```
bash> setserial /dev/ttyS1 uart none
bash> modprobe nsc-ircc dongle_id=0x09
bash> irattach irda0 -s
```

`dongle_id` depends on your IR hardware and can be found from your hardware documentation. As you did for SIR, take a look at */proc/net/irda/discovery* to see whether things are okay thus far. Sometimes, FIR communication hangs at higher speeds. If irdadump shows a communication freeze, either put on your kernel hacking hat and fix the code, or try lowering the negotiated speed by tweaking */proc/sys/net/irda/max_baud_rate*.

Note that unlike the Bluetooth physical layer that can establish one-to-many connections, IR can support only a single connection per physical device at a time.

Device Example: IR Dongle

Dongles are IR devices that plug into serial or USB ports. Some microcontrollers (such as Cirrus Logic's EP7211 shown in Figure 16.5) that have on-chip IR controllers wired to their UARTs are also considered dongles.

Dongle drivers are a set of control methods responsible for operations such as changing the communication speed. They have four entry points: `open()`, `reset()`, `change_speed()`, and `close()`. These entry points are defined as part of a `dongle_driver` structure and are invoked from the context of the IrDA kernel thread, *irda_sir_wq*. Dongle driver methods are allowed to block because they are invoked from process context with no locks held. The IrDA core offers three helper functions to dongle drivers: `sirdev_raw_write()` and `sirdev_raw_read()` to exchange control data with the associated UART, and `sirdev_set_dtr_rts()` to wiggle modem control lines connected to the UART.

Because you're probably more likely to add kernel support for dongles than modify other parts of Linux-IrDA, let's implement an example dongle driver. Assume that you're enabling a yet-unsupported simple serial IR dongle that communicates only at 19200 or 57600 baud. Assume also that when the user wants to toggle the baud rate between these two values, you have to hold the UART's *Request-to-Send* (RTS) pin low for 50 microseconds and pull it back high for 25 microseconds. Listing 16.2 implements a dongle driver for this device.

Listing 16.2. An Example Dongle Driver

Code View:

```
#include <linux/delay.h>
#include <net/irda/irda.h>
#include "sir-dev.h"   /* Assume that this sample driver lives in
                          drivers/net/irda/ */

/* Open Method. This is invoked when an irattach is issued on the
   associated UART */
static int
mydongle_open(struct sir_dev *dev)
{
  struct qos_info *qos = &dev->qos;

  /* Power the dongle by setting modem control lines, DTR/RTS. */
  sirdev_set_dtr_rts(dev, TRUE, TRUE);

  /* Speeds that mydongle can accept */
  qos->baud_rate.bits &= IR_19200|IR_57600;

  irda_qos_bits_to_value(qos); /* Set QoS */
  return 0;
}

/* Change baud rate */
static int
mydongle_change_speed(struct sir_dev *dev, unsigned speed)
{
  if ((speed == 19200) || (speed = 57600)){
    /* Toggle the speed by pulsing RTS low
       for 50 us and back high for 25 us */
    sirdev_set_dtr_rts(dev, TRUE, FALSE);
    udelay(50);
    sirdev_set_dtr_rts(dev, TRUE, TRUE);
    udelay(25);
    return 0;
  } else {
    return -EINVAL;
```

```
  }
}

/* Reset */
static int
mydongle_reset(struct sir_dev *dev)
{
  /* Reset the dongle as per the spec, for example,
     by pulling DTR low for 50 us */
  sirdev_set_dtr_rts(dev, FALSE, TRUE);
  udelay(50);
  sirdev_set_dtr_rts(dev, TRUE, TRUE);
  dev->speed = 19200; /* Reset speed is 19200 baud */
  return 0;
}

/* Close */
static int
mydongle_close(struct sir_dev *dev)
{
  /* Power off the dongle as per the spec,
     for example, by pulling DTR and RTS low.. */
  sirdev_set_dtr_rts(dev, FALSE, FALSE);
  return 0;
}

/* Dongle Driver Methods */
static struct dongle_driver mydongle = {
  .owner     = THIS_MODULE,
  .type      = MY_DONGLE,            /* Add this to the enumeration
                                        in include/linux/irda.h */
  .open      = mydongle_open,        /* Open */
  .reset     = mydongle_reset,       /* Reset */
  .set_speed = mydongle_change_speed, /* Change Speed */
  .close     = mydongle_close,       /* Close */
};

/* Initialize */
static int __init
mydongle_init(void)
{
  /* Register the entry points */
  return irda_register_dongle(&mydongle);
}
/* Release */
static void __exit
mydongle_cleanup(void)
{
  /* Unregister entry points */
  irda_unregister_dongle(&mydongle);
}

module_init(mydongle_init);
module_exit(mydongle_cleanup);
```

For real-life examples, look at *drivers/net/irda/tekram.c* and *drivers/net/irda/ep7211_ir.c*.

Now that you have the physical layer running, let's venture to look at IrDA protocols.

IrComm

IrComm emulates serial ports. Applications such as terminal emulators and protocols such as PPP can run unchanged over the virtual serial interfaces created by IrComm. IrComm is implemented by two related modules, *ircomm.ko* and *ircomm_tty.ko*. The former provides core protocol support, while the latter creates and manages the emulated serial port nodes */dev/ircommX*.

Networking

There are three ways to get TCP/IP applications running over IrDA:

1. Asynchronous PPP over IrComm
2. Synchronous PPP over IrNET
3. Ethernet emulation with IrLAN

Networking over IrComm is equivalent to running asynchronous PPP over a serial port, so there is nothing out of the ordinary in this scenario.

Asynchronous PPP needs to mark the start and end of frames using techniques such as byte stuffing, but if PPP is running over data links such as Ethernet, it need not be burdened with the overhead of a framing protocol. This is called synchronous PPP and is used to configure networking over IrNET.[3] Passage through the PPP layer provides features such as on-demand IP address configuration, compression, and authentication.

[3] For a scholarly discussion on networking over IrNET, read www.hpl.hp.com/personal/Jean_Tourrilhes/Papers/IrNET.Demand.html.

To start IrNET, insert *irnet.ko*. This also creates the character device node */dev/irnet*, which is a control channel over which you can attach the PPP daemon:

```
bash> pppd /dev/irnet 9600 noauth a.b.c.d:a.b.c.e
```

This yields the `pppX` network interfaces at either ends with the respective IP addresses set to a.b.c.d and a.b.c.e. The interfaces can now beam TCP/IP packets.

IrLAN provides raw Ethernet emulation over IrDA. To network your laptops using IrLAN, do the following at both ends:

- Insert *irlan.ko*. This creates the network interface, `irlanX`, where `X` is the assigned interface number.
- Configure the `irlanX` interfaces. To set the IP address, do this:

```
bash> ifconfig irlanX a.b.c.d
```

Or automate it by adding the following line to */etc/sysconfig/network-scripts/-ifcfg-irlan0*:[4]

[4] The location of this file is distribution-dependent.

```
DEVICE=irlanX IPADDR=a.b.c.d
```

You can now telnet between the laptops over the `irlanX` interfaces.

IrDA Sockets

To develop custom applications over IrDA, use the IrSock interface. To create a socket over TinyTP, do this:

```
int fd = socket(AF_IRDA, SOCK_STREAM, 0);
```

For a datagram socket over IrLMP, do this:

```
int fd = socket(AF_IRDA, SOCK_DGRAM, 0);
```

Look at the *irsockets/* directory in the *irda-utils* package for code examples.

Linux Infrared Remote Control

The goal of the LIRC project is to let you control your Linux computer via a remote. For example, you can use LIRC to control applications that play MP3 music or DVD movies via buttons on your remote. LIRC is architected into

1. A base LIRC module called *lirc_dev*.

2. A hardware-specific physical layer driver. IR hardware that interface via serial ports use *lirc_serial.* To allow *lirc_serial* to do its job without interference from the kernel serial driver, dissociate the latter as you did earlier for FIR:

   ```
   bash> setserial /dev/ttySX uart none
   ```

   You may have to replace *lirc_serial* with a more suitable low-level LIRC driver depending on your IR device.

3. A user mode daemon called *lircd* that runs over the low-level LIRC driver. Lircd decodes signals arriving from the remote and is the centerpiece of LIRC. Support for many remotes are implemented in the form of user-space drivers that are part of lircd. Lircd exports a UNIX-domain socket interface */dev/lircd* to higher applications. Connecting to lircd via */dev/lircd* is the key to writing LIRC-aware applications.

4. An LIRC mouse daemon called *lircmd* that runs on top of lircd. Lircmd converts messages from lircd to mouse events. These events can be read from a named pipe */dev/lircm* and input to programs such as *gpm* or X Windows.

5. Tools such as *irrecord* and *irsend*. The former records signals received from your remote and helps you generate IR configuration files for a new remote. The latter streams IR commands from your Linux machine.

Visit the LIRC home page hosted at www.lirc.org to download all these and to obtain insights on its design and usage.

> **IR Char Drivers**
>
> If your embedded device requires only simple Infrared receive capabilities, it might be using a miniaturized IR receiver (such as the TSOP1730 chip from Vishay Semiconductors). An example application device is an IR locator installed in hospital rooms to read data emitted by IR badges worn by nurses. In this scenario, the IrDA stack is not relevant because of the absence of IrDA protocol interactions. It may also be an overkill to port LIRC to the locator if it's using a lean proprietary protocol to parse received data. An easy solution might be to implement a tiny read-only char or misc driver that exports raw IR data to a suitable application via */dev* or */sys* interfaces.

Looking at the Sources

Look inside *drivers/net/irda/* for IrDA low-level drivers, *net/irda/* for the protocol implementation, and *include/net/irda/* for the header files. Experiment with *proc/sys/net/irda/** to tune the IrDA stack and explore */proc/net/irda/** for state information pertaining to different IrDA layers.

Table 16.3 contains the main data structures used in this section and their location in the source tree. Table 16.4 lists the main kernel programming interfaces that you used in this section along with the location of their definitions.

Table 16.3. Summary of Data Structures

Data Structure	Location	Description
`dongle_driver`	*drivers/net/irda/sir-dev.h*	Dongle driver entry points
`sir_dev`	*drivers/net/irda/sir-dev.h*	Representation of an SIR device
`qos_info`	*include/net/irda/qos.h*	Quality-of-Service information

Table 16.4. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`irda_register_dongle()`	*drivers/net/irda/sir_dongle.c*	Registers a dongle driver
`irda_unregister_dongle()`	*drivers/net/irda/sir_dongle.c*	Unregisters a dongle driver
`sirdev_set_dtr_rts()`	*drivers/net/irda/sir_dev.c*	Wiggles modem control lines on the serial port attached to the IR device
`sirdev_raw_write()`	*drivers/net/irda/sir_dev.c*	Writes to the serial port attached to the IR device

Kernel Interface	Location	Description
`sirdev_raw_read()`	*drivers/net/irda/sir_dev.c*	Reads from the serial port attached to the IR device

WiFi

WiFi, or *wireless local-area network* (WLAN), is an alternative to wired LAN and is generally used within a campus. The IEEE 802.11a WLAN standard uses the 5GHz ISM (*Industrial, Scientific, Medical*) band and supports speeds of up to 54Mbps. The 802.11b and the 802.11g standards use the 2.4GHz band and support speeds of 11Mbps and 54Mbps, respectively.

WLAN resembles wired Ethernet in that both are assigned MAC addresses from the same address pool and both appear to the operating system as regular network interfaces. For example, *Address Resolution Protocol* (ARP) tables contain WLAN MAC addresses alongside Ethernet MAC addresses.

WLAN and wired Ethernet differ significantly at the link layer, however:

- The 802.11 WLAN standard uses collision avoidance (CSMA/CA) rather than collision detection (CSMA/CD) used by wired Ethernet.
- WLAN frames, unlike Ethernet frames, are acknowledged.
- Due to security issues inherent in wireless networking, WLAN uses an encryption mechanism called *Wired Equivalent Privacy* (WEP) to provide a level of security equivalent to wired Ethernet. WEP combines a 40-bit or a 104-bit key with a random 24-bit initialization vector to encrypt and decrypt data.

WLAN supports two communication modes:

1. *Ad-hoc mode*, where a small group of nearby stations directly communicate without using an access point.
2. *Infrastructure mode*, where data exchanges pass via an access point. Access points periodically broadcast a *service set identifier* (SSID or ESSID) that identifies one WLAN network from another.

Let's find out how Linux supports WLAN.

Configuration

The *Wireless Extensions* project defines a generic Linux API to configure WLAN device drivers in a device-independent manner. It also provides a set of common tools to set and access information from WLAN drivers. Individual drivers implement support for Wireless Extensions to connect themselves with the common interface and, hence, with the tools.

With Wireless Extensions, there are primarily three ways to talk to WLAN drivers:

1. Standard operations using the *iwconfig* utility. To glue your driver to iwconfig, you need to implement prescribed functions corresponding to commands that set parameters such as ESSID and WEP keys.

2. Special-purpose operations using *iwpriv*. To use iwpriv over your driver, define private ioctls relevant to your hardware and implement the corresponding handler functions.

3. WiFi-specific statistics through */proc/net/wireless*. For this, implement the `get_wireless_stats()` method in your driver. This is in addition to the `get_stats()` method implemented by NIC drivers for generic statistics collection as described in the section "Statistics" in Chapter 15, "Network Interface Cards."

WLAN drivers tie these three pieces of information inside a structure called `iw_handler_def`, defined in *include/net/iw_handler.h*. The address of this structure is supplied to the kernel via the device's `net_device` structure (discussed in Chapter 15) during initialization. Listing 16.3 shows a skeletal WLAN driver implementing support for Wireless Extensions. The comments in the listing explain the associated code.

Listing 16.3. Supporting Wireless Extensions

Code View:

```
#include <net/iw_handler.h>
#include <linux/wireless.h>

/* Populate the iw_handler_def structure with the location and number
   of standard and private handlers, argument details of private
   handlers, and location of get_wireless_stats() */
static struct iw_handler_def mywifi_handler_def = {
  .standard           = mywifi_std_handlers,
  .num_standard       = sizeof(mywifi_std_handlers) /
                        sizeof(iw_handler),
  .private            = (iw_handler *) mywifi_pvt_handlers,
  .num_private        = sizeof(mywifi_pvt_handlers) /
                        sizeof(iw_handler),
  .private_args       = (struct iw_priv_args *)mywifi_pvt_args,
  .num_private_args   = sizeof(mywifi_pvt_args) /
                        sizeof(struct iw_priv_args),
  .get_wireless_stats = mywifi_stats,
};


/* Handlers corresponding to iwconfig */
static iw_handler mywifi_std_handlers[] = {
  NULL,               /* SIOCSIWCOMMIT */
  mywifi_get_name,    /* SIOCGIWNAME */
  NULL,               /* SIOCSIWNWID */
  NULL,               /* SIOCGIWNWID */
  mywifi_set_freq,    /* SIOCSIWFREQ */
  mywifi_get_freq,    /* SIOCGIWFREQ */
  mywifi_set_mode,    /* SIOCSIWMODE */
  mywifi_get_mode,    /* SIOCGIWMODE */
  /* ... */
};
#define MYWIFI_MYPARAMETER   SIOCIWFIRSTPRIV

/* Handlers corresponding to iwpriv */
static iw_handler mywifi_pvt_handlers[] = {
  mywifi_set_myparameter,
  /* ... */
};

/* Argument description of private handlers */
```

```
static const struct iw_priv_args mywifi_pvt_args[] = {
  { MYWIFI_MYPARAMATER,
    IW_PRIV_TYPE_INT | IW_PRIV_SIZE_FIXED | 1, 0, "myparam"},
}

struct iw_statistics mywifi_stats; /* WLAN Statistics */

/* Method to set operational frequency supplied via mywifi_std_handlers. Similarly
implement the rest of the methods */
mywifi_set_freq()
{
  /* Set frequency as specified in the data sheet */
  /* ... */
}

/* Called when you read /proc/net/wireless */
static struct iw_statistics *
mywifi_stats(struct net_device *dev)
{
  /* Fill the fields in mywifi_stats */
  /* ... */
  return(&mywifi_stats);
}


/*Device initialization. For PCI-based cards, this is called from the
  probe() method. Revisit init_mycard() in Listing 15.1 in Chapter 15
  for a full discussion */
static int
init_mywifi_card()
{
  struct net_device *netdev;
  /* Allocate WiFi network device. Internally calls
     alloc_etherdev() */
  netdev = alloc_ieee80211(sizeof(struct mywifi_priv));
  /* ... */

  /* Register Wireless Extensions support */
  netdev->wireless_handlers = &mywifi_handler_def;

  /* ... */
  register_netdev(netdev);
}
```

With Wireless Extensions support compiled in, you can use iwconfig to configure the ESSID and the WEP key, peek at supported private commands, and dump network statistics:

```
bash> iwconfig eth1 essid blue key 1234-5678-9012-3456-7890-1234-56

bash> iwconfig eth1
eth1   IEEE 802.11b ESSID:"blue" Nickname:"ipw2100"
       Mode:Managed Frequency:2.437 GHz Access Point: 00:40:96:5E:07:2E
       ...
```

```
       Encryption key:1234-5678-9012-3456-7890-1234-56
       Security mode:open
       ...
bash> dhcpcd eth1

bash> ifconfig
eth1    Link encap:Ethernet  Hwaddr 00:13:E8:02:EE:18
        inet addr:192.168.0.41   Bcasr:192.168.0.255
        Mask:255.255.255.0
       ...
bash> iwpriv eth1
eth1   Available private ioctls:
       myparam   (8BE2): set 2 int  & get 0

bash> cat /proc/net/wireless
Inter-| sta-| Quality        | Discarded packets               | Missed | WE
 face | tus |link level noise|nwid  crypt   frag  retry  misc| beacon | 19
  eth1: 0004  100.  207.  0.    0      0      0     2     1        0
```

Local iwconfig parameters such as the ESSID and WEP key should match the configuration at the access point.

There is another project called *cfg80211* having similar goals as Wireless Extensions. This has been merged into the mainline kernel starting with the 2.6.22 kernel release.

Device Drivers

There are hundreds of WLAN *original equipment manufacturers* (OEMs) in the market, and cards come in several form factors such as PCI, Mini PCI, CardBus, PCMCIA, Compact Flash, USB, and SDIO (see the sidebar "WiFi over SDIO") . However, the number of controller chips that lie at the heart of these devices, and hence the number of Linux device drivers, are relatively less in number. The Intersil Prism chipset, Lucent Hermes chipset, Atheros chipset, and Intel Pro/Wireless are among the popular WLAN controllers. The following are example devices built using these controllers:

- Intersil Prism2 WLAN Compact Flash Card— The Orinoco WLAN driver, part of the kernel source tree, supports both Prism-based and Hermes-based cards. Look at *orinoco.c* and *hermes.c* in *drivers/net/wireless/* for the sources. *orinoco_cs* provides PCMCIA/CF Card Services support.

- The Cisco Aironet CardBus adapter— This card uses an Atheros chipset. The *Madwifi* project (http://madwifi.org/) offers a Linux driver that works on hardware built using Atheros controllers. The Madwifi source base is not part of the kernel source tree primarily due to licensing issues. One of the modules of the Madwifi driver called *Hardware Access Layer* (HAL) is closed source. This is because the Atheros chip is capable of operating at frequencies that are outside permissible ISM bands and can work at various power levels. The U.S. *Federal Communications Commission* (FCC) mandates that such settings should not be easily changeable by users. Part of HAL is distributed as binary-only to comply with FCC regulations. This binary-only portion is independent of the kernel version.

- Intel Pro/Wireless Mini PCI (and PCIe Mini) cards embedded on many laptops— The kernel source tree contains drivers for these cards. The drivers for the 2100 and 2200 BG series cards are *drivers/net/wireless/ipw2100.c* and *drivers/net/wireless/ipw2200.c*, respectively. These devices need on-card firmware to work. You can download the firmware from http://ipw2100.sourceforge.net/ or http://ipw2200.sourceforge.net/ depending on whether you have a 2100 or a 2200. The section "Microcode Download" in Chapter 4, "Laying the Groundwork," described the steps needed to download

firmware on to these cards. Intel's distribution terms for the firmware are restrictive.

- WLAN USB devices— The Atmel USB WLAN driver (http://atmelwlandriver.sourceforge.net/) supports USB WLAN devices built using Atmel chipsets.

The WLAN driver's task is to let your card appear as a normal network interface. Driver implementations are generally split into the following parts:

1. The interface that communicates with the Linux networking stack— We discussed this in detail in the section "The Net Device Interface" in Chapter 15. You can use Listing 15.1 in that chapter as a template to implement this portion of your WLAN driver.

2. Form factor–specific code— If your card is a PCI card, it has to be architected to conform to the kernel PCI subsystem as described in Chapter 10, "Peripheral Component Interconnect." Similarly, PCMCIA and USB cards have to tie in with their respective core layers.

3. Chipset specific part— This is the cornerstone of the WLAN driver and is based on register specifications in the chip's data sheet. Many companies do not release adequate documentation for writing open source device drivers, however, so this portion of some Linux WLAN drivers is at least partly based on reverse-engineering.

4. Support for Wireless Extensions— Listing 16.3, shown earlier, implements an example.

Hardware-independent portions of the 802.11 stack are reusable across drivers, so they are implemented as a collection of common library functions in the *net/ieee80211/* directory. *ieee80211* is the core protocol module, but if you want to configure WEP keys via the `iwconfig` command, you have to load *ieee80211_crypt* and *ieee80211_crypt_wep*, too. To generate debugging output from the 802.11 stack, enable `CONFIG_IEEE80211_DEBUG` while configuring your kernel. You can use */proc/net/ieee80211/debug_level* as a knob to fine-tune the type of debug messages that you want to see. Starting with the 2.6.22 release, the kernel has an alternate 802.11 stack (*net/mac80211/*) donated by a company called Devicescape. WiFi device drivers may migrate to this new stack in the future.

WiFi over SDIO

Like PCMCIA cards whose functionality has extended from storage to various other technologies, SD cards are no longer confined to the consumer electronics memory space. The *Secure Digital Input/Output* (SDIO) standard brings technologies such as WiFi, Bluetooth, and GPS to the SD realm. The Linux-SDIO project hosted at http://sourceforge.net/projects/sdio-linux/ offers drivers for several SDIO cards.

Go to www.sdcard.org to browse the SD Card Association's website. The latest standards adopted by the association are microSD and miniSD, which are miniature form factor versions of the SD card.

Looking at the Sources

WiFi device drivers live in *drivers/net/wireless/*. Look inside *net/wireless/* for the implementations of Wireless Extensions and the new *cfg80211* configuration interface. The two Linux 802.11 stacks live under *net/ieee80211/* and *net/mac80211/*, respectively.

Cellular Networking

Global System for Mobile Communications (GSM) is a prominent digital cellular standard. GSM networks are called 2G or second-generation networks. GPRS represents the evolution from 2G to 2.5G. Unlike 2G networks, 2.5G networks are "always on." Compared to GSM's 9.6Kbps throughput, GPRS supports theoretical speeds of up to 170Kbps. 2.5G GPRS has given way to 3G networks based on technologies such as CDMA that offer higher speeds.

In this section, let's look at GPRS and CDMA.

GPRS

Because GPRS chips are cellular modems, they present a UART interface to the system and usually don't require specialized Linux drivers. Here's how Linux supports common GPRS hardware:

1. For a system with built-in GPRS support, say, a board having a Siemens MC-45 module wired to the microcontroller's UART channel, the conventional Linux serial driver can drive the link.

2. For a PCMCIA/CF GPRS device such as an Options GPRS card, *serial_cs*, the generic serial Card Services driver allows the rest of the operating system to see the card as a serial device. The first unused serial device (*/dev/ttySX*) gets allotted to the card. Look at Figure 9.5 in Chapter 9, for an illustration.

3. For USB GPRS modems, a USB-to-serial converter typically converts the USB port to a virtual serial port. The *usbserial* driver lets the rest of the system see the USB modem as a serial device (*/dev/ttyUSBX*). The section "USB-Serial" in Chapter 11 discussed USB-to-serial converters.

The above driver descriptions also hold for driving *Global Positioning System* (GPS) receivers and networking over GSM.

After the serial link is up, you may establish a network connection via `AT` commands, a standard language to talk to modems. Cellular devices support an extended `AT` command set. The exact command sequence depends on the particular cellular technology in use. Consider for example, the `AT` string to connect over GPRS. Before entering data mode and connecting to an external network via a *gateway GPRS support node* (GGSN), a GPRS device must define a context using an `AT` command. Here's an example context string:

```
'AT+CGDCONT=1,"IP","internet1.voicestream.com","0.0.0.0",0,0'
```

where 1 stands for a context number, `IP` is the packet type, `internet1.-voicestream.com` is an *access point name* (APN) specific to the service provider, and `0.0.0.0` asks the service provider to choose the IP address. The last two parameters pertain to data and header compression. A username and password are usually not needed.

As you saw in Chapter 9, PPP is used as the vehicle to carry TCP/IP payload over GPRS. A common syntax for invoking the PPP daemon, *pppd*, is this:

```
bash> pppd ttySX call connection-script
```

where `ttySX` is the serial port over which PPP runs, and `connection-script` is a file in */etc/ppp/peers/*[5] that contains the `AT` command sequence to establish the link. After establishing connection and completing authentication, PPP starts a *Network Control Protocol* (NCP) such as *Internet Protocol Control Protocol* (IPCP). When IPCP successfully negotiates IP addresses, PPP starts talking with the TCP/IP stack.

[5] The path name might vary depending on the distribution you use.

Here is an example PPP connection script (*/etc/ppp/peer/gprs-seq*) for connecting to a GPRS service provider at 57600 baud. For the semantics of all constituent lines in the script, refer to the man pages of *pppd*:

```
57600
connect "/usr/sbin/chat -s -v "" AT+CGDCONT=1,"IP",
"internet2.voicestream.com","0.0.0.0",0,0 OK AT+CGDATA="PPP",1"
crtscts
noipdefault
modem
usepeerdns
defaultroute
connect-delay 3000
```

CDMA

For performance reasons, many CDMA PC Cards have an internal USB controller through which a CDMA modem is connected. When such cards are inserted, the system sees one or more new PCI-to-USB bridges on the PCI bus. Let's take the example of a Huawei CDMA CardBus card. Look at the additional entries in the *lspci* output after inserting this card into the CardBus slot of a laptop:

Code View:

```
bash> lspci -v
...
07:00:0 USB Controller: NEC Corporation USB (rev 43) (prog-if 10 [OHCI])
07:00:1 USB Controller: NEC Corporation USB (rev 43) (prog-if 10 [OHCI])
07:00:2 USB Controller: NEC Corporation USB 2.0 (rev 04) (prog-if 20 [EHCI])
```

These are standard OHCI and EHCI controllers, so the host controller drivers on Linux seamlessly talk to them. If a CDMA card, however, uses a host controller unsupported by the kernel, you will have the unenviable task of writing a new USB host controller driver. Let's take a closer look at the new USB buses in the above *lspci* output and see whether we can find any devices connected to them:

Code View:

```
bash> cat /proc/bus/usb/devices
T:  Bus=07 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480 MxCh= 2
B:  Alloc=  0/800 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=01 MxPS=64 #Cfgs=  1
...
T:  Bus=06 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12  MxCh= 1
B:  Alloc=  0/900 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 1.10 Cls=09(hub  ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
...
T:  Bus=05 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12  MxCh= 1
B:  Alloc=  0/900 us ( 0%), #Int=  1, #Iso=  0
```

```
D:  Ver= 1.10 Cls=09(hub  ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
...
T:  Bus=05 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  3 Spd=12  MxCh= 0
D:  Ver= 1.01 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=16 #Cfgs=  1
P:  Vendor=12d1 ProdID=1001 Rev= 0.00
S:  Manufacturer=Huawei Technologies
S:  Product=Huawei Mobile
C:* #Ifs= 2 Cfg#= 1 Atr=e0 MxPwr=100mA
I:  If#= 0 Alt= 0 #EPs= 3 Cls=ff(vend.) Sub=ff Prot=ff Driver=pl2303
E:  Ad=81(I) Atr=03(Int.) MxPS=  16 Ivl=128ms
E:  Ad=8a(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=0b(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
I:  If#= 1 Alt= 0 #EPs= 2 Cls=ff(vend.) Sub=ff Prot=ff Driver=pl2303
E:  Ad=83(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=06(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
...
```

The top three entries (`bus7`, `bus6`, and `bus5`) correspond to the three host controllers present in the CDMA card. The last entry shows that a full-speed (12Mbps) USB device is connected to bus 5. This device has a `vendorID` of `0x12d1` and a `productID` of `0x1001`. As is evident from the preceding output, the USB core has bound this device to the *pl2303* driver. If you look at the source file of the PL2303 Prolific USB-to-serial adapter driver (*drivers/usb/serial/pl2303.c*), you will find the following member in the `usb_device_id` table:

```
static struct usb_device_id id_table [] = {
  /* ... */
  {USB_DEVICE(HUAWEI_VENDOR_ID, HUAWEI_PRODUCT_ID)},
  /* ... */
};
```

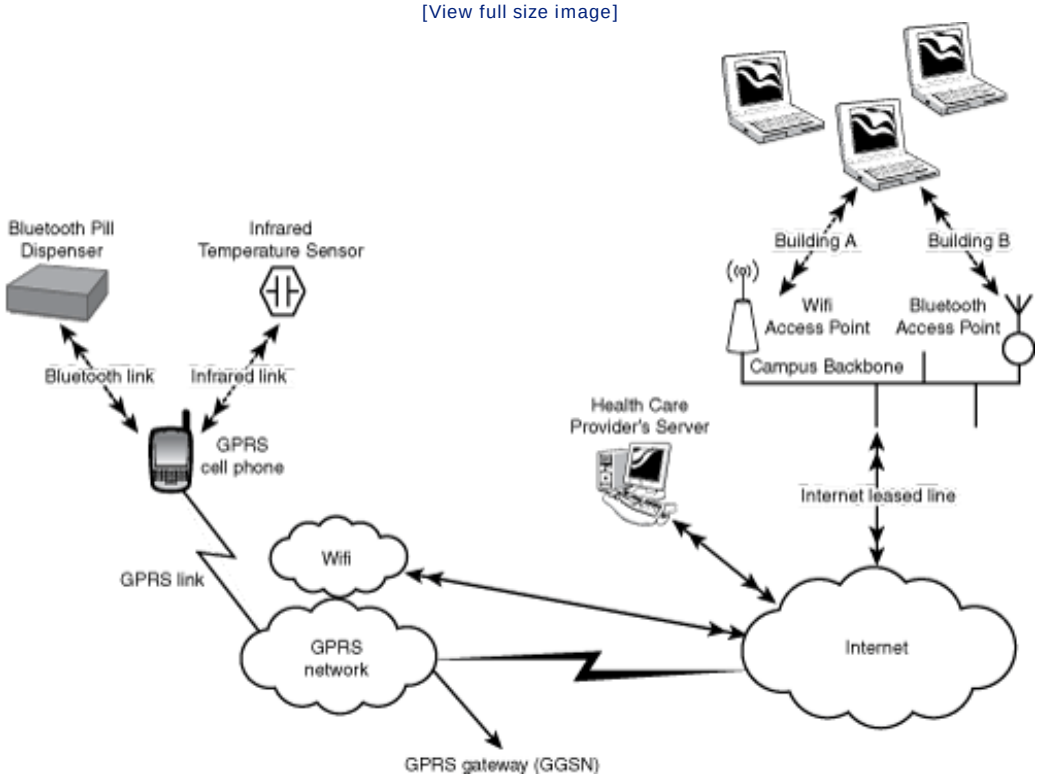
A quick peek at *pl2303.h* living in the same directory confirms that `HUAWEI_VENDOR_ID` and `HUAWEI_PRODUCT_ID` match the values that you just gleaned from */proc/bus/usb/devices.* The pl2303 driver presents a serial interface, */dev/ttyUSB0*, over the detected USB-to-serial converter. You can send `AT` commands to the CDMA modem over this interface. Attach *pppd* over this device and connect to the net. You are now a 3G surfer!

Current Trends

At one end of today's on-the-move connectivity spectrum, there are standards that allow coupling between cellular networks and WiFi to provide cheaper networking solutions. At the other end, technologies such as Bluetooth and Infrared are being integrated into GPRS cell phones to bridge consumer electronics devices with the Internet. Figure 16.7 shows a sample scenario.

Figure 16.7. Coupling between wireless technologies.

[View full size image]



In tandem with the coupling of existing standards and technologies, there is a steady stream of new communication standards arriving in the wireless space.

Zigbee (www.zigbee.org) adopts the new 802.15.4 standard for wireless networking in the embedded space that is characterized by low range, speed, energy consumption, and code footprint. It primarily targets home and industrial automation. Of the wireless protocols discussed in this chapter, Zigbee is closest to Bluetooth but is considered complementary rather than competitive with it.

WiMax (*Worldwide interoperability for Microwave access*), based on the IEEE 802.16 standard, is a *metropolitan-area network* (MAN) flavor of WiFi that has a range of several kilometers. It supports fixed connectivity for homes and offices, and a mobile version for networking on the go. WiMax is a cost-effective way to solve the last-mile connectivity problem (which is analogous to the task of reaching your home from the nearest metro rail station) and create broadband clouds that span large areas. The WiMax forum is hosted at www.wimaxforum.org.

MIMO (*Multiple In Multiple Out*) is a new multiple-antenna technology utilized by WiFi and WiMax products to enhance their speed, range, and connectivity.

Working groups are developing new standards that fall under the ambit of fourth-generation or 4G networking. 4G will signal the convergence of several communication technologies.

Some of the new communication technologies are transparent to the operating system and work unchanged with existing drivers and protocol stacks. Others such as Zigbee need new drivers and protocol stacks but do not have accepted open source implementations yet. Linux mirrors the state of the art, so look out for support for these new standards in future kernel releases.

Chapter 17. Memory Technology Devices

In This Chapter

- What's Flash Memory? 504
- Linux-MTD Subsystem 505
- Map Drivers 506
- NOR Chip Drivers 511
- NAND Chip Drivers 513
- User Modules 516
- MTD-Utils 518
- Configuring MTD 519
- eXecute In Place 520
- The Firmware Hub 520
- Debugging 524
- Looking at the Sources 524

When you push the power switch on your handheld, it's more than likely that it boots from flash memory. When you click some buttons to save data on your cell phone, in all probability, your data starts life in flash memory. Today, Linux has penetrated the embedded space and is no longer confined to desktops and servers. Linux avatars manifest in PDAs, music players, set-top boxes, and even medical-grade devices. The *Memory Technology Devices* (MTD) subsystem of the kernel is responsible for interfacing your system with various flavors of flash memory found in these devices. In this chapter, let's use the example of a Linux handheld to learn about MTD.

What's Flash Memory?

Flash memory is rewritable storage that does not need power supply to hold information. Flash memory banks are usually organized into *sectors.* Unlike conventional storage, writes to flash addresses have to be preceded by an erase of the corresponding locations. Moreover, erases of portions of flash can be performed only at the granularity of individual sectors. Because of these constraints, flash memory is best used with device drivers and filesystems that are tailored to suit them. On Linux, such specially designed drivers and filesystems are provided by the MTD subsystem.

Flash memory chips generally come in two flavors: NOR and NAND. *NOR* is the variety used to store firmware images on embedded devices, whereas *NAND* is used for large, dense, cheap, but imperfect[1] storage as required by solid-state mass storage media such as USB pen drives and *Disk-On-Modules* (DOMs). NOR flash chips are connected to the processor via address and data lines like normal RAM, but NAND flash chips are interfaced using I/O and control lines. So, code resident on NOR flash can be executed in place, but that stored on NAND flash has to be copied to RAM before execution.

[1] It's normal to have bad blocks scattered across NAND flash regions as you will learn in the section, "NAND Chip Drivers."

Chapter 17. Memory Technology Devices

In This Chapter

- What's Flash Memory? 504
- Linux-MTD Subsystem 505
- Map Drivers 506
- NOR Chip Drivers 511
- NAND Chip Drivers 513
- User Modules 516
- MTD-Utils 518
- Configuring MTD 519
- eXecute In Place 520
- The Firmware Hub 520
- Debugging 524
- Looking at the Sources 524

When you push the power switch on your handheld, it's more than likely that it boots from flash memory. When you click some buttons to save data on your cell phone, in all probability, your data starts life in flash memory. Today, Linux has penetrated the embedded space and is no longer confined to desktops and servers. Linux avatars manifest in PDAs, music players, set-top boxes, and even medical-grade devices. The *Memory Technology Devices* (MTD) subsystem of the kernel is responsible for interfacing your system with various flavors of flash memory found in these devices. In this chapter, let's use the example of a Linux handheld to learn about MTD.

What's Flash Memory?

Flash memory is rewritable storage that does not need power supply to hold information. Flash memory banks are usually organized into *sectors.* Unlike conventional storage, writes to flash addresses have to be preceded by an erase of the corresponding locations. Moreover, erases of portions of flash can be performed only at the granularity of individual sectors. Because of these constraints, flash memory is best used with device drivers and filesystems that are tailored to suit them. On Linux, such specially designed drivers and filesystems are provided by the MTD subsystem.

Flash memory chips generally come in two flavors: NOR and NAND. *NOR* is the variety used to store firmware images on embedded devices, whereas *NAND* is used for large, dense, cheap, but imperfect[1] storage as required by solid-state mass storage media such as USB pen drives and *Disk-On-Modules* (DOMs). NOR flash chips are connected to the processor via address and data lines like normal RAM, but NAND flash chips are interfaced using I/O and control lines. So, code resident on NOR flash can be executed in place, but that stored on NAND flash has to be copied to RAM before execution.

[1] It's normal to have bad blocks scattered across NAND flash regions as you will learn in the section, "NAND Chip Drivers."

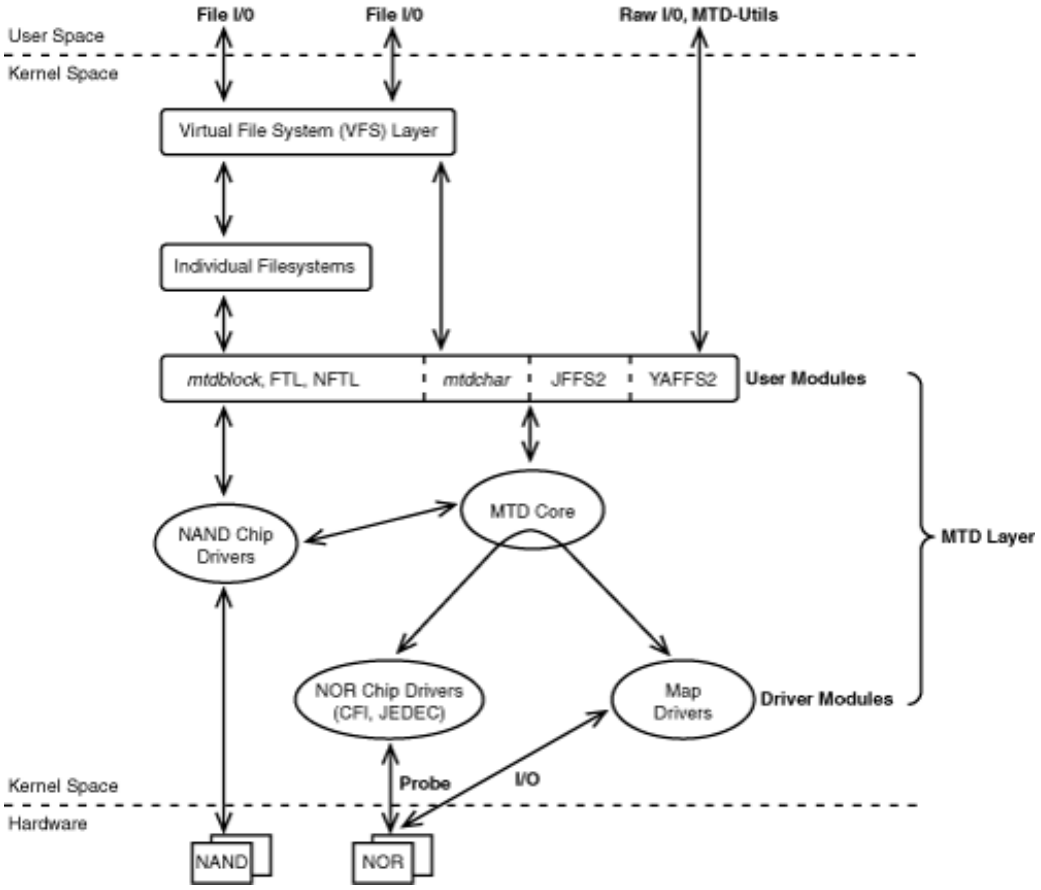
Linux-MTD Subsystem

The kernel's MTD subsystem shown in Figure 17.1 provides support for flash and similar nonvolatile solid-state storage. It consists of the following:

- The *MTD core*, which is an infrastructure consisting of library routines and data structures used by the rest of the MTD subsystem
- *Map drivers* that decide what the processor ought to do when it receives requests for accessing the flash
- *NOR Chip drivers* that know about commands required to talk to NOR flash chips
- *NAND Chip drivers* that implement low-level support for NAND flash controllers
- *User Modules*, the layer that interacts with user-space programs
- Individual device drivers for some special flash chips

Figure 17.1. The Linux-MTD subsystem.

[View full size image]


File I/0
File I/0
Raw I/0, MTD-Utils
User Space
Kernel Space
Virtual File System (VFS) Layer
Individual Filesystems
mtdblock, FTL, NFTL
mtdchar
JFFS2
YAFFS2
User Modules
NAND Chip Drivers
MTD Core
MTD Layer
NOR Chip Drivers (CFI, JEDEC)
Map Drivers
Driver Modules
Probe
I/O
Kernel Space
Hardware
NAND
NOR

Map Drivers

To MTD-enable your device, your first task is to tell MTD how to access the flash device. For this, you have to map your flash memory range for CPU access and provide methods to operate on the flash. The next task is to inform MTD about the different storage partitions residing on your flash. Unlike hard disks on PC-compatible systems, flash-based storage does not contain a standard partition table on the media. Because of this, disk-partitioning tools such as *fdisk* and *cfdisk*[2] cannot be used to partition flash devices. Instead, partitioning information has to be implemented as part of kernel code.[3] These tasks are accomplished with the help of an MTD *map driver*.

[2] *Fdisk* and *cfdisk* are used to manipulate the partition table residing in the first hard disk sector on PC systems.

[3] You may also pass partitioning information to MTD via the kernel command line argument `mtdpart=`, if you enable `CONFIG_MTD_CMDLINE_PARTS` during kernel configuration. Look at *drivers/mtd/cmdlinepart.c* for the usage syntax.

To better understand the function of map drivers, let's look at an example.

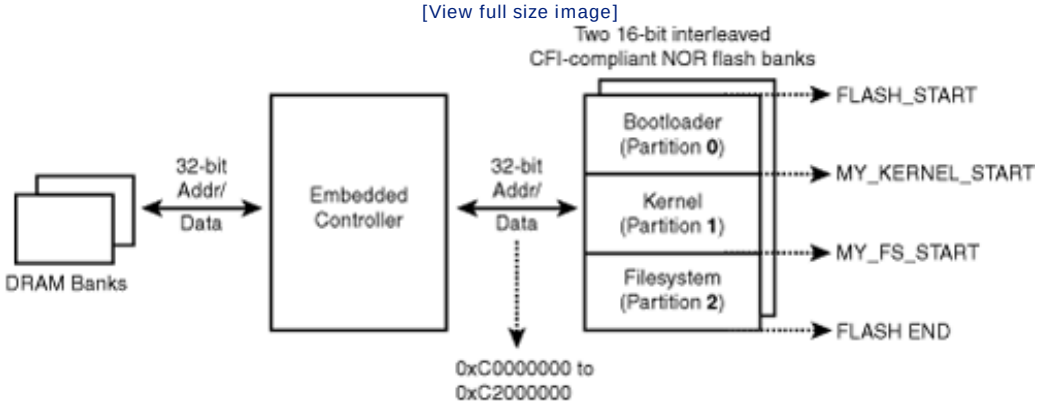
Device Example: Handheld

Consider the Linux handheld shown in Figure 17.2. The flash has a size of 32MB and is mapped to `0xC0000000` in the processor's address space. It contains three partitions, one each for the bootloader, the kernel, and the root filesystem. The bootloader partition starts from the top of the flash, the kernel partition begins at offset `MY_KERNEL_START`, and the root filesystem starts at offset `MY_FS_START`.[4] The bootloader and the kernel reside on *read-only* partitions to avoid unexpected damage, while the filesystem partition is flagged *read-write*.

[4] Some devices have additional partitions for bootloader parameters, extra filesystems, and recovery kernels.

Figure 17.2. Flash Memory on a sample Linux handheld.

[View full size image]



Let's first create the flash map and then proceed with the driver initialization. The map driver has to translate

the flash layout shown in the figure to an `mtd_partition` structure. Listing 17.1 contains the `mtd_partition` definition corresponding to Figure 17.2. Note that the `mask_flags` field holds the permissions to be masked, so `MTD_WRITEABLE` implies a read-only partition.

Listing 17.1. Creating an MTD Partition Map

```
Code View:
#define FLASH_START          0x00000000
#define MY_KERNEL_START      0x00080000 /* 512K for bootloader */
#define MY_FS_START          0x00280000 /* 2MB for kernel */
#define FLASH_END            0x02000000 /* 32MB */
static struct mtd_partition pda_partitions[] = {
  {
    .name       = "pda_btldr",        /* This string is used by
                                         /proc/mtd to identify
                                         the bootloader partition */
    .size:      = (MY_KERNEL_START-FLASH_START),
    .offset     = FLASH_START,        /* Start from top of flash */
    .mask_flags = MTD_WRITEABLE       /* Read-only partition */
  },
  {
    .name       = "pda_krnl",         /* Kernel partition */
    .size:      = (MY_FS_START-MY_KERNEL_START),
    .offset     = MTDPART_OFS_APPEND, /* Start immediately after
                                         the bootloader partition */
    .mask_flags = MTD_WRITEABLE       /* Read-only partition */
  },
  {
    .name:      = "pda_fs",           /* Filesystem partition */
    .size:      = MTDPART_SIZ_FULL,   /* Use up the rest of the
                                         flash */
    .offset     = MTDPART_OFS_NEXTBLK,/* Align this partition with
                                         the erase size */
  }
};
```

Listing 17.1 uses `MTDPART_OFS_APPEND` to start a partition adjacent to the previous one. The start addresses of writeable partitions, however, need to be aligned with the erase/sector size of the flash chip. To achieve this, the filesystem partition uses `MTD_OFS_NEXTBLK` rather than `MTD_OFS_APPEND`.

Now that you have populated the `mtd_partition` structure, let's proceed and complete a basic map driver for the example handheld. Listing 17.2 registers the map driver with the MTD core. It's implemented as a platform driver, assuming that your architecture-specific code registers an associated platform device having the same name. Rewind to the section "Device Example: Cell Phone" in Chapter 6, "Serial Drivers," for a discussion on platform devices and platform drivers. The `platform_device` is defined by the associated architecture-specific code as follows:

```
struct resource pda_flash_resource = { /* Used by Listing 17.3 */
  .start = 0xC0000000,                 /* Physical start of the
                                          flash in Figure 17.2 */
  .end   = 0xC0000000+0x02000000-1,    /* Physical end of flash */
  .flags = IORESOURCE_MEM,             /* Memory resource */
};
struct platform_device pda_platform_device = {
```

```
  .name = "pda",                    /* Platform device name */
  .id   = 0,                        /* Instance number */
  /* ... */
  .resource = &pda_flash_resource, /* See above */
};
platform_device_register(&pda_platform_device);
```

Listing 17.2. Registering the Map Driver

```
static struct platform_driver pda_map_driver = {
  .driver = {
    .name     =  "pda",         /* ID */
   },
  .probe      =  pda_mtd_probe, /* Probe */
  .remove     =  NULL,          /* Release */
  .suspend    =  NULL,          /* Power management */
  .resume     =  NULL,          /* Power management */
};

/* Driver/module Initialization */
static int __init pda_mtd_init(void)
{
  return platform_driver_register(&pda_map_driver);
}

/* Module Exit */
static int __init pda_mtd_exit(void)
{
  return platform_driver_uregister(&pda_map_driver);
}
```

Because the kernel finds that the name of the platform driver registered in Listing 17.2 matches with that of an already-registered platform device, it invokes the probe method, `pda_mtd_probe()`, shown in Listing 17.3. This routine

- Reserves the flash memory address range using `request_mem_region()`, and obtains CPU access to that memory using `ioremap_nocache()`. You learned how to do this in Chapter 10, "Peripheral Component Interconnect."
- Populates a `map_info` structure (discussed next) with information such as the start address and size of flash memory. The information in this structure is used while performing the probing in the next step.
- Probes the flash via a suitable MTD chip driver (discussed in the next section). Only the chip driver knows how to query the chip and elicit the command-set required to access it. The chip layer tries different permutations of bus widths and interleaves while querying. In Figure 17.2, two 16-bit flash banks are connected in parallel to fill the 32-bit processor bus width, so you have a two-way interleave.
- Registers the `mtd_partition` structure that you populated earlier, with the MTD core.

Before looking at Listing 17.3, let's meet the `map_info` structure. It contains the address, size, and width of the flash memory and routines to access it:

```
struct map_info {
  char * name;               /* Name */
  unsigned long size;        /* Flash size */
  int bankwidth;             /* In bytes */
  /* ... */
  /* You need to implement custom routines for the following methods
     only if you have special needs. Else populate them with built-
     in methods using simple_map_init() as done in Listing 17.3 */
  map_word (*read)(struct map_info *, unsigned long);
  void     (*write)(struct map_info *, const map_word,
                    unsigned long);
  /* ... */
};
```

While we are in the topic of accessing flash chips, let's briefly revisit memory barriers that we discussed in Chapter 4, "Laying the Groundwork." An instruction reordering that appears semantically unchanged to the compiler (or the processor) may not be so in reality, so the ordering of data operations on flash memory is best left alone. You don't want to, for example, end up erasing a flash sector after writing to it, instead of doing the reverse. Also, the same flash chips, and hence their device drivers, are used on diverse embedded processors having different instruction reordering algorithms. For these reasons, MTD drivers are notable users of hardware memory barriers. `simple_map_write()`, a generic routine available to map drivers for use as the `write()` method in the `map_info` structure previously listed, inserts a call to `mb()` before returning. This ensures that the processor does not reorder flash reads or writes across the barrier.

Listing 17.3. Map Driver Probe Method

Code View:

```
#include <linux/mtd/mtd.h>
#include <linux/mtd/map.h>
#include <linux/ioport.h>

static int
pda_mtd_probe(struct platform_device *pdev)
{
  struct map_info *pda_map;
  struct mtd_info *pda_mtd;
  struct resource *res = pdev->resource;

  /* Populate pda_map with information obtained
     from the associated platform device */
  pda_map->virt = ioremap_nocache(res->start,
                                  (res->end – res->start + 1));
  pda_map->name = pdev->dev.bus_id;
  pda_map->phys = res->start;
  pda_map->size = res->end – res->start + 1;
  pda_map->bankwidth = 2;     /* Two 16-bit banks sitting
                                 on a 32-bit bus */
  simple_map_init(&pda_map);  /* Fill in default access methods */

  /* Probe via the CFI chip driver */
  pda_mtd = do_map_probe("cfi_probe", &pda_map);
  /* Register the mtd_partition structure */
  add_mtd_partitions(pda_mtd, pda_partitions, 3); /* Three Partitions */

  /* ... */
}
```

Don't worry if the CFI probing done in Listing 17.3 seems esoteric. It's discussed in the next section when we look at NOR chip drivers.

MTD now knows how your flash device is organized and how to access it. When you boot the kernel with your map driver compiled in, user-space applications can respectively see your bootloader, kernel, and filesystem partitions as */dev/mtd/0*, */dev/mtd/1*, and */dev/mtd/2*. So, to test drive a new kernel image on the handheld, you can do this:

```
bash> dd if=zImage.new of=/dev/mtd/1
```

Flash Partitioning from Bootloaders

The Redboot bootloader maintains a partition table that holds flash layout, so if you are using Redboot on your embedded device, you can configure your flash partitions in the bootloader instead of writing an MTD map driver. To ask MTD to parse flash mapping information from Redboot's partition table, turn on `CONFIG_MTD_REDBOOT_PARTS` during kernel configuration.

NOR Chip Drivers

As you might have noticed, the NOR flash chip used by the handheld in Figure 17.2 is labeled *CFI-compliant.* CFI stands for *Common Flash Interface*, a specification designed to do away with the need for developing separate drivers to support chips from different vendors. Software can query CFI-compliant flash chips and automatically detect block sizes, timing parameters, and the command-set to be used for communication. Drivers that implement specifications such as CFI and JEDEC are called *chip drivers*.

According to the CFI specification, software must write `0x98` to location `0x55` within flash memory to initiate a query. Look at Listing 17.4 to see how MTD implements CFI query.

Listing 17.4. Querying CFI-compliant Flash

```
/* Snippet from cfi_probe_chip() (2.6.23.1 kernel) defined in
   drivers/mtd/chips/cfi_probe.c, with comments added */

/* cfi is a pointer to struct cfi_private defined in
   include/linux/mtd/cfi.h */

/* ... */

/* Ask the device to enter query mode by sending
   0x98 to offset 0x55 */
cfi_send_gen_cmd(0x98, 0x55, base, map, cfi,
                 cfi->device_type, NULL);

/* If the device did not return the ASCII characters
   'Q', 'R' and 'Y', the chip is not CFI-compliant */
if (!qry_present(map, base, cfi)) {
  xip_enable(base, map, cfi);
  return 0;
}

/* Elicit chip parameters and the command-set, and populate
   the cfi structure */
if (!cfi->numchips) {
  return cfi_chip_setup(map, cfi);
}
/* ... */
```

The CFI specification defines various command-sets that compliant chips can implement. Some of the common ones are as follows:

- Command-set `0001`, supported by Intel and Sharp flash chips
- Command-set `0002`, implemented on AMD and Fujitsu flash chips
- Command-set `0020`, used on ST flash chips

MTD supports these command-sets as kernel modules. You can enable the one supported by your flash chip via the kernel configuration menu.

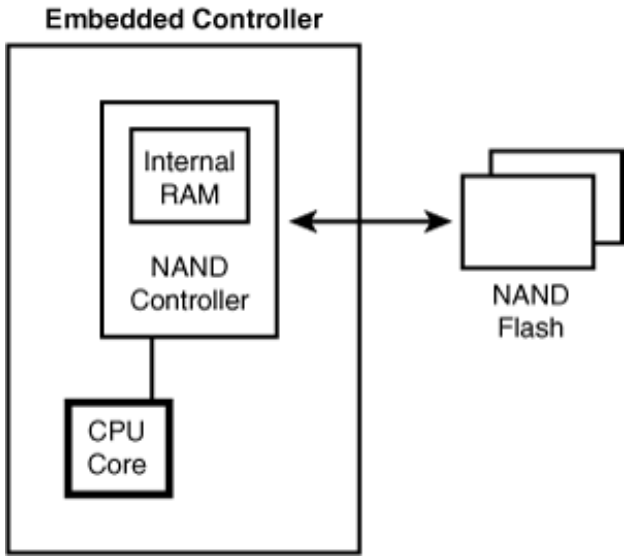
NAND Chip Drivers

NAND technology users such as USB pen drives, DOMs, Compact Flash memory, and SD/MMC cards emulate standard storage interfaces such as SCSI or IDE over NAND flash, so you don't need to develop NAND drivers to communicate with them.[5] On-board NAND flash chips need special drivers, however, and are the topic of this section.

[5] Unless you are writing drivers for the storage media itself. If you are embedding Linux on a device that will export part of its NAND partition to the outside world as a USB mass storage device, you do have to contend with NAND drivers.

As you learned previously in this chapter, NAND flash chips, unlike their NOR counterparts, are not connected to the CPU via data and address lines. They interface to the CPU through special electronics called a *NAND flash controller* that is part of many embedded processors. To read data from NAND flash, the CPU issues an appropriate *read* command to the NAND controller. The controller transfers data from the requested flash location to an internal RAM memory, also part of the controller. The data transfer is done in units of the flash chip's *page size* (for example, 2KB). In general, the denser the flash chip, the larger is its page size. Note that the page size is different from the flash chip's *block size*, which is the minimum erasable flash memory unit (for example, 16KB). After the transfer operation completes, the CPU reads the requested NAND contents from the internal RAM. Writes to NAND flash are done similarly, except that the controller transfers data from the internal RAM to flash. The connection diagram of NAND flash memory on an embedded device is shown in Figure 17.3.

Figure 17.3. NAND flash connection.



Because of this unconventional mode of addressing, you need special drivers to work with NAND storage. MTD provides such drivers to manage NAND-resident data. If you are using a supported chip, you have to enable only the appropriate low-level MTD NAND driver. If you are writing a NAND flash driver, however, you need to explore two datasheets: the NAND flash controller and the NAND flash chip.

NAND flash chips do not support automatic configuration using protocols such as CFI. You have to manually inform MTD about the properties of your NAND chip by adding an entry to the `nand_flash_ids[]` table defined

in *drivers/mtd/nand/nand_ids.c*. Each entry in the table consists of an identifier name, the device ID, page size, erase block size, chip size, and options such as the bus width.

There is another characteristic that goes hand in hand with NAND memory. NAND flash chips, unlike NOR chips, are not faultless. It's normal to have some problem bits and bad blocks scattered across NAND flash regions. To handle this, NAND devices associate a *spare area* with each flash page (for example, 64 bytes of spare area for each 2KB data page). The spare area contains *out-of-band* (OOB) information to help perform bad block management and error correction. The OOB area includes *error correcting codes* (ECCs) to implement error correction and detection. ECC algorithms correct single-bit errors and detect multibit errors. The `nand_ecclayout` structure defined in *include/mtd/mtd-abi.h* specifies the layout of the OOB spare area:

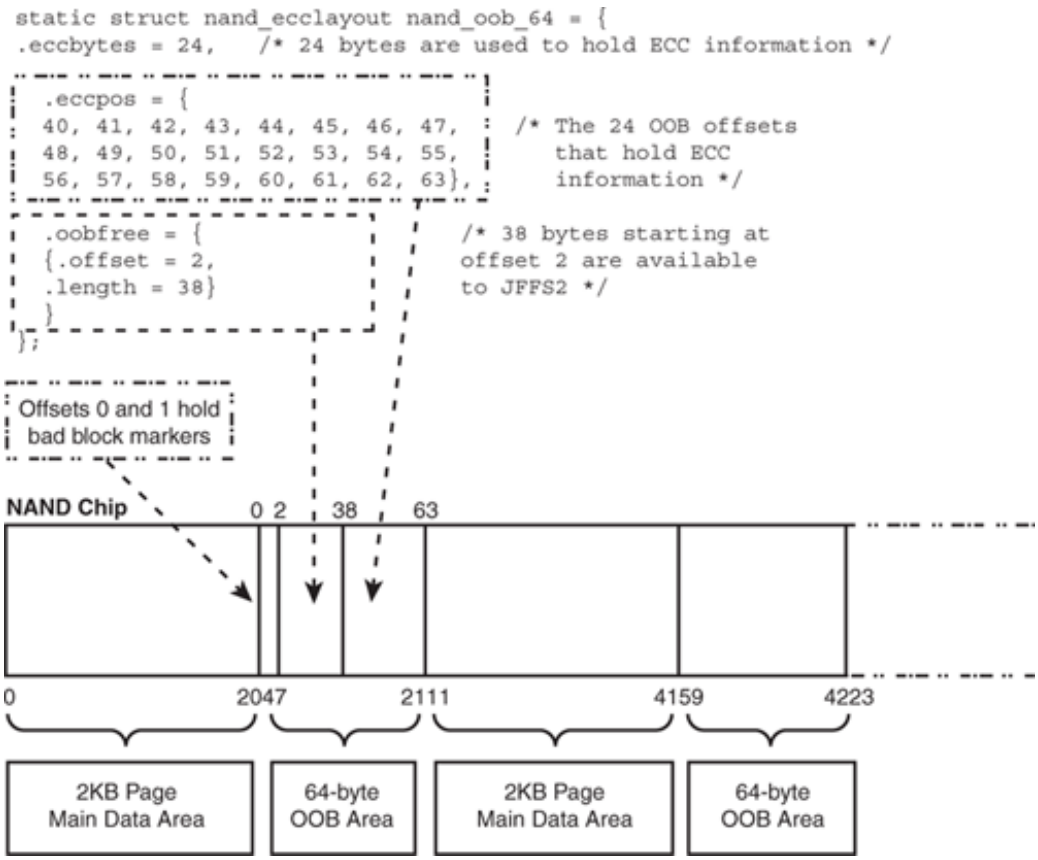
```
struct nand_ecclayout {
  uint 32_t eccbytes;
  uint32_t  eccpos[64];
  uint32_t  oobavail;
  struct    nand_oobfree oobfree[MTD_MAX_OOBFREE_ENTRIES];
};
```

In this structure, `eccbytes` holds the number of OOB bytes that store ECC data, and `eccpos` is an array of offsets into the OOB area that contains the ECC data. `oobfree` records the unused bytes in the OOB area available to flash filesystems for storing flags such as *clean markers* that signal successful completion of erase operations.

Individual NAND drivers initialize their `nand_ecclayout` according to the chip's properties. Figure 17.4 illustrates the layout of a NAND flash chip having a page size of 2KB. The OOB semantics used by the figure is the default for 2KB page-sized chips as defined in the generic NAND driver, *drivers/mtd/nand/nand_base.c*.

Figure 17.4. Layout of a NAND flash chip.

[View full size image]



Often, the NAND controller performs error correction and detection in hardware by operating on the ECC fields in the OOB area. If your NAND controller does not support error management, however, you will need to get MTD to do that for you in software. The MTD *nand_ecc* driver (*drivers/mtd/nand/nand_ecc.c*) implements software ECC.

Figure 17.4 also shows OOB memory bytes that contain bad block markers. These markers are used to flag faulty flash blocks and are usually present in the OOB region belonging to the first page of each block. The position of the marker inside the OOB area depends on the properties of the chip. Bad block markers are either set at the factory during manufacture, or by software when it detects wear in a block. MTD implements bad block management in *drivers/mtd/nand/nand_bbt.c*.

The `mtd_partition` structure used in Listing 17.1 for the NOR flash in Figure 17.2 works for NAND memory, too. After you MTD-enable your NAND flash, you can access the constituent partitions using standard device nodes such as */dev/mtd/X* and */dev/mtdblock/X*. If you have a mix of NOR and NAND memories on your hardware, *X* can be either a NOR or a NAND partition. If you have a total of more than 32 flash partitions, accordingly change the value of `MAX_MTD_DEVICES` in *include/linux/mtd/mtd.h*.

To effectively make use of NAND storage, you need to use a filesystem tuned for NAND access, such as JFFS2 or YAFFS2, in tandem with the low-level NAND driver. We discuss these filesystems in the next section.

User Modules

After you have added a map driver and chosen the right chip driver, you're all set to let higher layers use the flash. User-space applications that perform file I/O need to view the flash device as if it were a disk, whereas programs that desire to accomplish raw I/O access the flash as if it were a character device. The MTD layer that achieves these and more is called *User Modules*, as shown in Figure 17.1. Let's look at the components constituting this layer.

Block Device Emulation

The MTD subsystem provides a block driver called *mtdblock* that emulates a hard disk over flash memory. You can put any filesystem, say EXT2, over the emulated flash disk. Mtdblock hides complicated flash access procedures (such as preceding a write with an erase of the corresponding sector) from the filesystem. Device nodes created by mtdblock are named */dev/mtdblock/X*, where *X* is the partition number. To create an EXT2 filesystem on the pda_fs partition of the handheld, as shown in Figure 17.2, do the following:

```
bash> mkfs.ext2 /dev/mtdblock/2     ⟶  Create an EXT2 filesystem
                                        on the second partition
bash> mount /dev/mtdblock/2 /mnt    ⟶  Mount the partition
```

As you will soon see, it's a much better idea to use JFFS2 rather than EXT2 to hold files on flash filesystem partitions.

The *File Translation Layer* (FTL) and the *NAND File Translation Layer* (NFTL) perform a transformation called *wear leveling.* Flash memory sectors can withstand only a finite number of erase operations (in the order of 100,000). Wear leveling prolongs flash life by distributing memory usage across the chip. Both FTL and NFTL provide device interfaces similar to *mtdblock* over which you can put normal filesystems. The corresponding device nodes are named */dev/nftl/X*, where *X* is the partition number. Certain algorithms used in these modules are patented, so there could be restrictions on usage.

Char Device Emulation

The mtdchar driver presents a linear view of the underlying flash device, rather than the block-oriented view required by filesystems. Device nodes created by mtdchar are named */dev/mtd/X*, where *X* is the partition number. You may update the bootloader partition of the handheld as shown in Figure 17.2, by using dd over the corresponding mtdchar interface:

```
bash> dd if=bootloader.bin of=/dev/mtd/0
```

An example use of a raw mtdchar partition is to hold POST error logs generated by the bootloader on an embedded device. Another use of a char flash partition on an embedded system is to store information similar to that present in the CMOS or the EEPROM on PC-compatible systems. This includes the boot order, power-on password, and *Vital Product Data* (VPD) such as the device serial number and model number.

JFFS2

Journaling Flash File System (JFFS) is considered the best-suited filesystem for flash memory. Currently, version 2 (JFFS2) is in use, and JFFS3 is under development. JFFS was originally written for NOR flash chips, but support for NAND devices is merged with the 2.6 kernel.

Normal Linux filesystems are designed for desktop computers that are shut down gracefully. JFFS2 is designed

for embedded systems where power failure can occur abruptly, and where the storage device can tolerate only a finite number of erases. During flash erase operations, current sector contents are saved in RAM. If there is a power loss during the slow erase process, entire contents of that sector can get lost. JFFS2 circumvents this problem using a log-structured design. New data is appended to a log that lives in an erased region. Each JFFS2 node contains metadata to track disjoint file locations. Memory is periodically reclaimed using garbage collection. Because of this design, flash writes do not have to go through a save-erase-write cycle, and this improves power-down reliability. The log-structure also increases flash life span by spreading out writes.

To create a JFFS2 image of a tree living under */path/to/filesystem/* on a flash chip having an erase size of 256KB, use *mkfs.jffs2* as follows:

```
bash> mkfs.jffs2 -e 256KiB –r /path/to/filesystem/ -o jffs2.img
```

JFFS2 includes a *garbage collector* (GC) that reclaims flash regions that are no longer in use. The garbage collection algorithm depends on the erase size, so supplying an accurate value makes it more efficient. To obtain the erase size of your flash partitions, you may seek the help of */proc/mtd.* The output for the Linux handheld shown in Figure 17.2 is as follows:

```
bash> cat /proc/mtd
dev:    size    erasesize   name
mtd0: 00100000  00040000  "pda_btldr"
mtd1: 00200000  00040000  "pda_krnl"
mtd2: 01400000  00040000  "pda_fs"
```

JFFS2 supports compression. Enable appropriate options under `CONFIG_JFFS2_COMPRESSION_OPTIONS` to choose available compressors, and look at *fs/jffs2/compr*.c* for their implementations.

Note that JFFS2 filesystem images are usually created on the host machine where you do cross-development and then transferred to the desired flash partition on the target device via a suitable download mechanism such as serial port, USB, or NFS. More on this in Chapter 18, "Embedding Linux."

YAFFS2

The implementation of JFFS2 in the 2.6 kernel includes features to work with the limitations of NAND flash, but *Yet Another Flash File System* (YAFFS) is a filesystem that is designed to function under constraints specific to NAND memory. YAFFS is not part of the mainline kernel, but some embedded distributions prepatch their kernels with support for YAFFS2, the current version of YAFFS.

You can download YAFFS2 source code and documentation from www.yaffs.net.

MTD-Utils

The MTD-utils package, downloadable from ftp://ftp.infradead.org/pub/mtd-utils/, contains several useful tools that work on top of MTD-enabled flash memory. Examples of included utilities are *flash_eraseall*, *nanddump*, *nandwrite*, and *sumtool*.

To erase the second flash partition (on NOR or NAND devices), use flash_eraseall as follows:

```
bash> flash_eraseall –j /dev/mtd/2
```

Because NAND chips may contain bad blocks, use ECC-aware programs such as nandwrite and nanddump to copy raw data, instead of general-purpose utilities, such as dd. To store the JFFS2 image that you created previously, on to the second NAND partition, do this:

```
bash> nandwrite /dev/mtd/2 jffs2.img
```

You can reduce JFFS2 mount times by inserting summary information into a JFFS2 image using sumtool and turning on `CONFIG_JFFS2_SUMMARY` while configuring your kernel. To write a summarized JFFS2 image to the previous NAND flash, do this:

```
bash> sumtool –e 256KiB –i jffs2.img –o jffs2.summary.img
bash> nandwrite /dev/mtd/2 jffs2.summary.img
bash> mount –t jffs2 /dev/mtdblock/2 /mnt
```

Configuring MTD

To MTD-enable your kernel, you have to choose the appropriate configuration options. For the flash chip shown in Figure 17.2, the required options are as follows:

```
CONFIG_MTD=y                → Enable the MTD subsystem
CONFIG_MTD_PARTITIONS=y     → Support for multiple partitions
CONFIG_MTD_GEN_PROBE=y      → Common routines for chip probing
CONFIG_MTD_CFI=y            → Enable CFI chip driver
CONFIG_MTD_PDA_MAP=y        → Option to enable the map driver
CONFIG_JFFS2_FS=y           → Enable JFFS2
```

`CONFIG_MTD_PDA_MAP` is assumed to be a new option added to enable the map driver we previously wrote. Each of these features can also be built as a kernel module unless you have an MTD-resident root filesystem. To mount the filesystem partition in Figure 17.2 as the root device during boot, ask your bootloader to append `root=/dev/mtdblock/2` to the command-line string that it passes to the kernel.

You may reduce kernel footprint by eliminating redundant probing. Because our example handheld has two parallel 16-bit banks sitting on a 32-bit physical bus (thus resulting in a two-way interleave and a 2-byte bank width), you can optimize using these additional options:

```
CONFIG_MTD_CFI_ADV_OPTIONS=y
CONFIG_MTD_CFI_GEOMETRY=y
CONFIG_MTD_MAP_BANK_WIDTH_2=y
CONFIG_MTD_CFI_I2=y
```

`CONFIG_MTD_MAP_BANK_WIDTH_2` enables a CFI bus width of 2, and `CONFIG_MTD_CFI_I2` sets an interleave of 2.

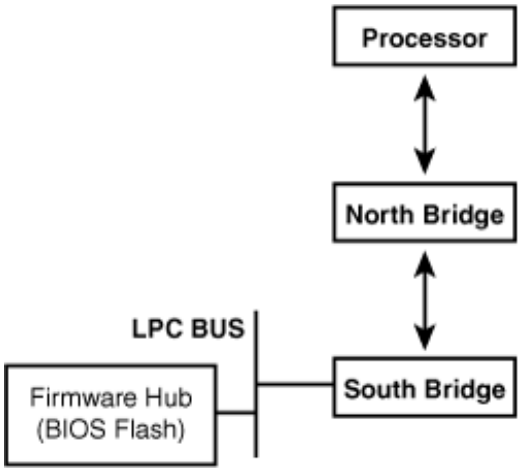
eXecute In Place

With *eXecute In Place* (XIP), you can run the kernel directly from flash. Because you do away with the extra step of copying the kernel to RAM, your kernel boots faster. The downside is that your flash memory requirement increases because the kernel has to be stored uncompressed. Before deciding to go the XIP route, also be aware that the slower instruction fetch times from flash can impact runtime performance.

The Firmware Hub

PC-compatible systems use a NOR flash chip called the *Firmware Hub* (FWH) to hold the BIOS. The FWH is not directly connected to the processor's address and data bus. Instead, it's interfaced via the *Low Pin Count* (LPC) bus, which is part of South Bridge chipsets. The connection diagram is shown in Figure 17.5.

Figure 17.5. The Firmware Hub on a PC-compatible system.



The MTD subsystem includes drivers to interface the processor with the FWH. FWHs are usually not compliant with the CFI specification. Instead, they conform to the JEDEC (*Joint Electron Device Engineering Council*) standard. To inform MTD about a yet unsupported JEDEC chip, add an entry to the `jedec_table` array in *drivers/mtd/chips/jedec_probe.c* with information such as the chip manufacturer ID and the command-set ID. Here is an example:

```
static const struct amd_flash_info jedec_table[] = {
  /* ... */
  {
    .mfr_id  = MANUFACTURER_ID, /* E.g.: MANUFACTURER_ST */
    .dev_id  = DEVICE_ID,       /* E.g.: M50FW080 */
    .name    = "MYNAME",        /* E.g.: "M50FW080" */
    .uaddr   = {
      [0] = MTD_UADDR_UNNECESSARY,
    },
    .DevSize  = SIZE_1MiB,    /* E.g.: 1MB */
    .CmdSet   = CMDSET,       /* Command-set to communicate with the
                                 flash chip e.g., P_ID_INTEL_EXT */
    .NumEraseRegions = 1,     /* One region */
    .regions = {
      ERASEINFO (0x10000, 16),/* Sixteen 64K sectors */
    }
  },
  /* ... */
};
```

When you have your chip details imprinted in the `jedec_table` as shown here, MTD should recognize your flash, provided you have enabled the right kernel configuration options. The following configuration makes the kernel aware of an FWH that interfaces to the processor via an Intel ICH2 or ICH4 South Bridge chipset:

```
CONFIG_MTD=y                  → Enable the MTD subsystem
CONFIG_MTD_GEN_PROBE=y        → Common routines for chip probing
CONFIG_MTD_JEDECPROBE=y       → JEDEC chip driver
CONFIG_MTD_CFI_INTELEXT=y     → The command-set for communicating
                                with the chip
CONFIG_MTD_ICHXROM=y          → The map driver
```

`CONFIG_MTD_JEDECPROBE` enables the JEDEC MTD chip driver, and `CONFIG_MTD_ICH2ROM` adds the MTD map driver that maps the FWH to the processor's address space. In addition, you need to include the appropriate command-set implementation (for example, `CONFIG_MTD_CFI_INTELEXT` for Intel Extension commands).

After these modules have been loaded, you can talk to the FWH from user-space applications via device nodes exported by MTD. You can, for example, reprogram the BIOS from user space using a simple application, as shown in Listing 17.5. Be warned that incorrectly operating this program can corrupt the BIOS and render your system unbootable!

Listing 17.5 operates on the MTD char device associated with the FWH, which it assumes to be */dev/mtd/0*. The program issues three MTD-specific *ioctl* commands:

- MEMUNLOCK to unlock the flash sectors prior to programming
- MEMERASE to erase flash sectors prior to rewriting
- MEMLOCK to relock the sectors after programming

Listing 17.5. Updating the BIOS

Code View:

```
#include <linux/mtd/mtd.h>
#include <stdio.h>
#include <fcntl.h>
#include <asm/ioctl.h>
#include <signal.h>
#include <sys/stat.h>

#define BLOCK_SIZE    4096
#define NUM_SECTORS   16
#define SECTOR_SIZE   64*1024

int
main(int argc, char *argv[])
{
  int fwh_fd, image_fd;
  int usect=0, lsect=0, ret;
  struct erase_info_user fwh_erase_info;
  char buffer[BLOCK_SIZE];
```

```
  struct stat statb;
  /* Ignore SIGINTR(^C) and SIGSTOP (^Z), lest
     you end up with a corrupted flash and an
     unbootable system */
  sigignore(SIGINT);
  sigignore(SIGTSTP);

  /* Open MTD char device */
  fwh_fd = open("/dev/mtd/0", O_RDWR);
  if (fwh_fd < 0) exit(1);

  /* Open BIOS image */
  image_fd = open("bios.img", O_RDONLY);
  if (image_fd < 0) exit(2);

  /* Sanity check */
  fstat(image_fd, &statb);
  if (statb.st_size != SECTOR_SIZE*NUM_SECTORS) {
    printf("BIOS image looks bad, exiting.\n");
    exit(3);
  }

  /* Unlock and erase all sectors */
  while (usect < NUM_SECTORS) {
    printf("Unlocking & Erasing Sector[%d]\r", usect+1);

    fwh_erase_info.start = usect*SECTOR_SIZE;
    fwh_erase_info.length = SECTOR_SIZE;

    ret = ioctl(fwh_fd, MEMUNLOCK, &fwh_erase_info);
    if (ret != 0) goto bios_done;

    ret = ioctl(fwh_fd, MEMERASE, &fwh_erase_info);
    if (ret != 0) goto bios_done;
    usect++;
  }

   /* Read blocks from the BIOS image and dump it to the
      Firmware Hub */
  while ((ret = read(image_fd, buffer, BLOCK_SIZE)) != 0) {
    if (ret < 0) goto bios_done;
    ret = write(fwh_fd, buffer, ret);
    if (ret <= 0) goto bios_done;
  }
  /* Verify by reading blocks from the BIOS flash and comparing
     with the image file */

  /* ... */

bios_done:

  /* Lock back the unlocked sectors */
  while (lsect < usect) {
    printf("Relocking Sector[%d]\r", lsect+1);

    fwh_erase_info.start = lsect*SECTOR_SIZE;
    fwh_erase_info.length = SECTOR_SIZE;

    ret = ioctl(fwh_fd, MEMLOCK, &fwh_erase_info);
```

```
    if (ret != 0) printf("Relock failed on sector %d!\n", lsect);
    lsect++;
  }

  close(image_fd);
  close(fwh_fd);

}
```

Debugging

To debug flash-related problems, enable `CONFIG_MTD_DEBUG` (*Device Drivers* → *Memory Technology Devices* → *Debugging*) during kernel configuration. You can further tune the debug verbosity level to between 0 and 3.

The Linux-MTD project page www.linux-mtd.infradead.org has FAQs, various pieces of documentation, and a *Linux-MTD JFFS HOWTO* that provides insights into JFFS2 design. The linux-mtd mailing list is the place to discuss questions related to MTD device drivers. Look at http://lists.infradead.org/pipermail/linux-mtd/ for the mailing list archives.

Looking at the Sources

In the kernel tree, the *drivers/mtd/* directory contains the sources for the MTD layer. Map, chip, and NAND drivers live in the *drivers/mtd/maps/*, *drivers/mtd/chips/*, and *drivers/mtd/nand/* subdirectories, respectively. Most MTD data structures are defined in header files present in *include/linux/mtd/*.

To access an unsupported BIOS firmware hub from Linux, implement a driver using *drivers/mtd/maps/ichxrom.c* as your starting point.

For examples of operating on NAND OOB data from user space, look at *nanddump.c* and *nandwrite.c* in the MTD-utils package.

Table 17.1 contains the main data structures used in this chapter and their location in the source tree. Table 17.2 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

Table 17.1. Summary of Data Structures

Data Structure	Location	Description
`mtd_partition`	*include/linux/mtd/partitions.h*	Representation of a flash chip's partition layout.
`map_info`	*include/linux/mtd/map.h*	Low-level access routines implemented by the map driver are passed to the chip driver using this structure.
`mtd_info`	*include/linux/mtd/mtd.h*	General device-specific information.
`erase_info`, `erase_info_user`	*include/linux/mtd/mtd.h*, *include/mtd/mtd-abi.h*	Structures used for flash erase management.
`cfi_private`	*include/linux/mtd/cfi.h*	Device-specific information maintained by NOR chip drivers.
`amd_flash_info`	*drivers/mtd/chips/jedec_probe.c*	Device-specific information supplied to the JEDEC chip driver.
`nand_ecclayout`	*include/mtd/mtd-abi.h*	Layout of the OOB spare area of a NAND chip.

Table 17.2. Summary of Kernel Programming Interfaces

Kernel Interface	Location	Description
`simple_map_init()`	*drivers/mtd/maps/map_funcs.c*	Initializes a `map_info` structure with generic flash access methods
`do_map_probe()`	*drivers/mtd/chips/chipreg.c*	Probes the NOR flash via a chip driver
`add_mtd_partitions()`	*drivers/mtd/mtdpart.c*	Registers an `mtd_partition` structure with the MTD core

Chapter 18. Embedding Linux

In This Chapter

- Challenges 528
- Component Selection 530
- Tool Chains 531
- Embedded Bootloaders 531
- Memory Layout 535
- Kernel Porting 537
- Embedded Drivers 538
- The Root Filesystem 544
- Test Infrastructure 548
- Debugging 548

Linux is making inroads into industry domains such as consumer electronics, telecom, networking, defense, and health care. With its popularity surging in the embedded space, it's more likely that you will use your Linux device driver skills to enable embedded devices rather than legacy systems. In this chapter, let's enter the world of embedded Linux wearing the lens of a device driver developer. Let's look at the software components of a typical embedded Linux solution and see how the device classes that you saw in the previous chapters tie in with common embedded hardware.

Challenges

Embedded systems present several significant software challenges:

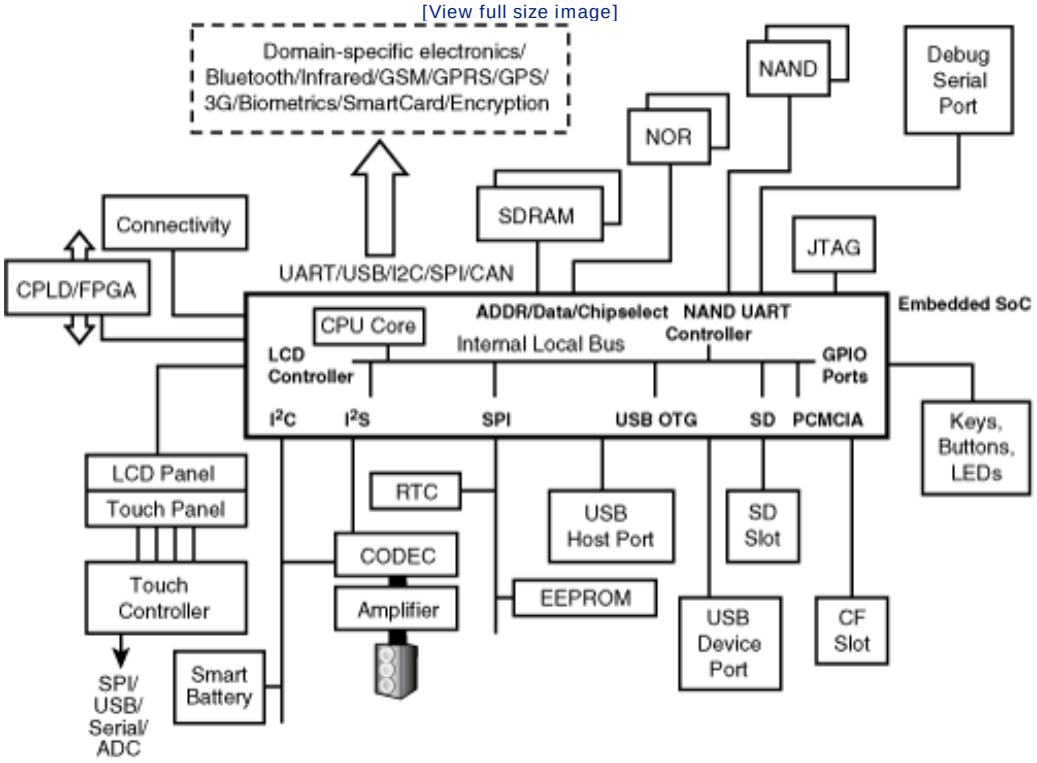
- Embedded software has to be cross-compiled and then downloaded to the target device to be tested and verified.
- Embedded systems, unlike PC-compatible computers, do not have fast processors, fat caches, and wholesome storage.
- It's often difficult to get mature development and debug tools for embedded hardware for free.
- The Linux community has a lot more experience on the x86 platform, so you are less likely to get instant online help from experts if you working on embedded computers.
- The hardware evolves in stages. You may have to start software development on a proof-of-concept prototype or a reference board, and progressively move on to engineering-level debug hardware and a few passes of production-level units.

All these result in a longer development cycle.

From a device-driver perspective, embedded software developers often face interfaces not commonly found on conventional computers. Figure 18.1 (which is an expanded version of Figure 4.2 in Chapter 4, "Laying the Groundwork") shows a hypothetical embedded device that could be a handheld, smart phone, *point-of-sale* (POS) terminal, kiosk, navigation system, gaming device, telemetry gadget on an automobile dashboard, IP phone, music player, digital set-top box, or even a pacemaker programmer. The device is built around an SoC and has some combination of flash memory, SDRAM, LCD, touch screen, USB OTG, serial ports, audio codec, connectivity, SD/MMC controller, Compact Flash, I^2C devices, SPI devices, JTAG, biometrics, smart card interfaces, keypad, LEDs, switches, and electronics specific to the industry domain. Modifying and debugging drivers for some of these devices can be tougher than usual: NAND flash drivers have to handle problems such as bad blocks and failed bits, unlike standard IDE storage drivers. Flash-based filesystems such as JFFS2, are more complex to debug than EXT2 or EXT3 filesystems. A USB OTG driver is more involved than a USB OHCI driver. The SPI subsystem on the kernel is not as mature as, say, the serial layer. Moreover, the industry domain using the embedded device might impose specific requirements such as quick response times or fast boot.

Figure 18.1. Block diagram of a hypothetical embedded device.

[View full size image]



Chapter 18. Embedding Linux

In This Chapter

- Challenges 528
- Component Selection 530
- Tool Chains 531
- Embedded Bootloaders 531
- Memory Layout 535
- Kernel Porting 537
- Embedded Drivers 538
- The Root Filesystem 544
- Test Infrastructure 548
- Debugging 548

Linux is making inroads into industry domains such as consumer electronics, telecom, networking, defense, and health care. With its popularity surging in the embedded space, it's more likely that you will use your Linux device driver skills to enable embedded devices rather than legacy systems. In this chapter, let's enter the world of embedded Linux wearing the lens of a device driver developer. Let's look at the software components of a typical embedded Linux solution and see how the device classes that you saw in the previous chapters tie in with common embedded hardware.

Challenges

Embedded systems present several significant software challenges:

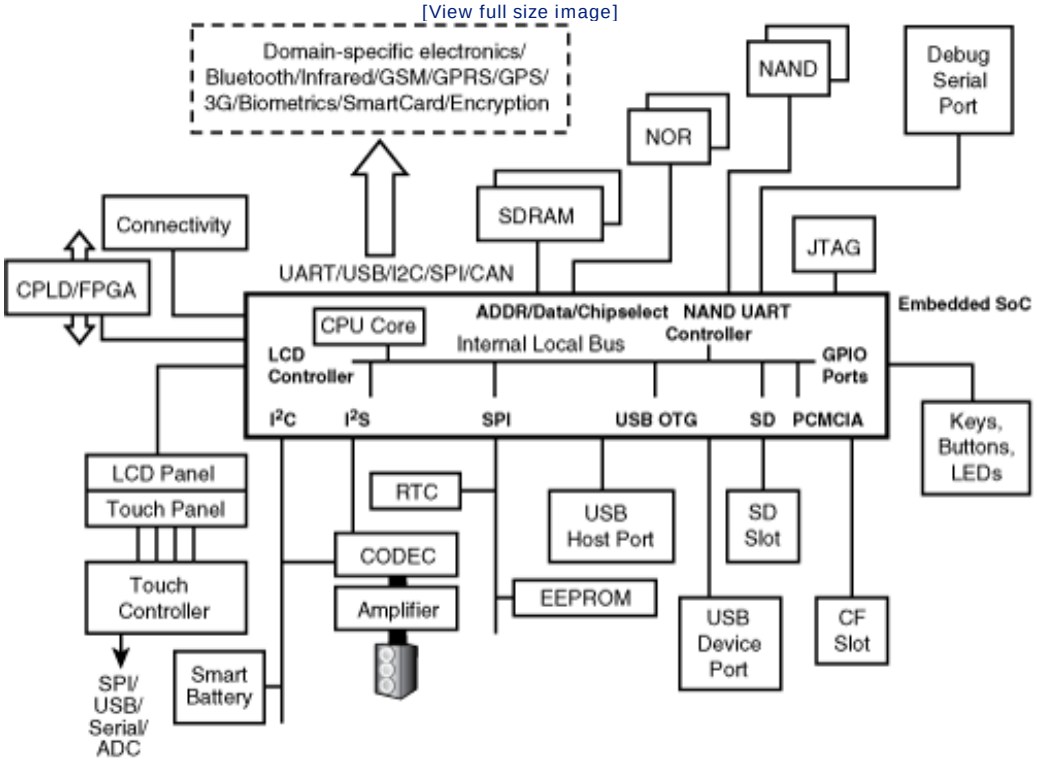
- Embedded software has to be cross-compiled and then downloaded to the target device to be tested and verified.
- Embedded systems, unlike PC-compatible computers, do not have fast processors, fat caches, and wholesome storage.
- It's often difficult to get mature development and debug tools for embedded hardware for free.
- The Linux community has a lot more experience on the x86 platform, so you are less likely to get instant online help from experts if you working on embedded computers.
- The hardware evolves in stages. You may have to start software development on a proof-of-concept prototype or a reference board, and progressively move on to engineering-level debug hardware and a few passes of production-level units.

All these result in a longer development cycle.

From a device-driver perspective, embedded software developers often face interfaces not commonly found on conventional computers. Figure 18.1 (which is an expanded version of Figure 4.2 in Chapter 4, "Laying the Groundwork") shows a hypothetical embedded device that could be a handheld, smart phone, *point-of-sale* (POS) terminal, kiosk, navigation system, gaming device, telemetry gadget on an automobile dashboard, IP phone, music player, digital set-top box, or even a pacemaker programmer. The device is built around an SoC and has some combination of flash memory, SDRAM, LCD, touch screen, USB OTG, serial ports, audio codec, connectivity, SD/MMC controller, Compact Flash, I^2C devices, SPI devices, JTAG, biometrics, smart card interfaces, keypad, LEDs, switches, and electronics specific to the industry domain. Modifying and debugging drivers for some of these devices can be tougher than usual: NAND flash drivers have to handle problems such as bad blocks and failed bits, unlike standard IDE storage drivers. Flash-based filesystems such as JFFS2, are more complex to debug than EXT2 or EXT3 filesystems. A USB OTG driver is more involved than a USB OHCI driver. The SPI subsystem on the kernel is not as mature as, say, the serial layer. Moreover, the industry domain using the embedded device might impose specific requirements such as quick response times or fast boot.

Figure 18.1. Block diagram of a hypothetical embedded device.

[View full size image]



Component Selection

Evaluating and selecting components is one of the important tasks undertaken during the concept phase of a project. Look at the sidebar "Choosing a Processor and Peripherals" for some important factors that hardware designers and product managers consider while choosing components for building an embedded device. In today's world, where time to market is often the critical factor driving device design, the software engineer also has a considerable say in shaping component selection. Availability of a Linux distribution can influence processor choice, while existence of device drivers or close starting points can affect the choice of peripheral chipsets.

Although the kernel engineer needs to do due diligence and evaluate several Linux distributions (or even operating systems), he may nix a technologically superior distribution in favor of a familiar one if he believes that'll mitigate project risks. Or a preferred distribution might be the one that offers indemnification from lawsuits arising out of kernel bugs, if that is a crucial consideration in the relevant industry domain. The electrical engineer can limit evaluation to processors supported by the chosen distribution and prefer peripheral chipsets enabled by the distribution in question.

Choosing a Processor and Peripherals

Let's look at some common questions that electrical engineers and product managers ask when selecting components for an embedded device. Assume that a hypothetical processor *P* is on the shortlist because it satisfies basic product requirements such as power consumption and packaging. *P* and accompanying peripheral chipsets are under evaluation:

Performance: Is the processor frequency sufficient to drive target applications? If the embedded device intends to implement CPU-intensive tasks, does the MIPS budgeting for all software subsystems balance with the processor's MIPS rating? If the target device requires high-resolution imaging, for example, will the MHz impact of graphics manipulation drag down the performance of other subsystems, such as networking?

Cost: Will I save a buck on the component but end up spending two more on the surrounding electronics? For example, will *P* need an extra regulator? Will I need to throw in an additional accessory, for example, an RTC chip, because *P* does not have one built-in? Does *P* have more pins than other processors under evaluation leading to a denser board having a larger number of layers and vias that increase the raw board cost? Does *P* consume more power and generate more heat necessitating a bigger power supply and additional passive components? Is there errata in the data sheet that has the possibility of increasing software development costs?

Functionality: What's the maximum size of DRAM, SRAM, NOR, and NAND memory that *P* can address?

Business Planning: Does *P*'s vendor offer an upgrade path to a higher horsepower processor that is a drop-in (pin-compatible) replacement? Is the vendor company stable?

Supplier: Is this a single-source component? If so, is the supplier volatile? What are the lead times to procure the parts?

End-of-Life: Is *P* likely to go end-of-life before the expected lifespan of the embedded device?

Credibility: Is *P* an accepted component? Do peripheral chipsets under evaluation have an industry segment behind them? Perhaps a landscape LCD under consideration is being used on automobile dashboards?

Ruggedness: Need the components be MIL (military) or industrial grade?

One has to evaluate different candidates and figure out the sweet spot in terms of all these.

Tool Chains

Because the target device is unlikely to be binary-compatible with your host development platform, you have to cross-compile embedded software using *tool chains.* Setting up a full-fledged tool chain entails building the following:

1. The GNU C (cross-)Compiler. GCC supports all platforms that Linux runs on, but you have to configure and build it to generate code for your target architecture. Essentially, you have to compile the compiler and generate the appropriate cross-compiler.

2. *Glibc*, the set of C libraries that you will need when you build applications for the target device.

3. *Binutils*, which includes the cross-assembler, and tools such as *objdump*.

Getting a development tool chain in place used to be a daunting task several years ago but is usually straightforward today because Linux distributions offer precompiled binaries and easy-installation tools for a variety of architectures.

Embedded Bootloaders

Bootloader development is usually the starting point of any embedded software effort. You have to decide whether to write a bootloader from scratch or tailor an existing open source bootloader to suit your needs. Each candidate bootloader might be built based on a different philosophy: small footprint, easy portability, fast boot, or the capability to support certain specific features. After you home-in on a starting point, you can design and implement device-specific modifications.

In this section, let's use the term *bootloader* to mean the boot suite. This includes the following:

- The BIOS, if present
- Any bootstrap code needed to put the bootloader onto the boot device
- One or more stages[1] of the actual bootloader

  [1] In embedded bootloader parlance, the first stage of a two-stage bootloader is sometimes called the *Initial Program Loader* (IPL), and the second stage is called the *Secondary Program Loader* (SPL).

- Any program executing on an external host machine that talks with the bootloader for the purpose of downloading firmware onto the target device

At the minimum, a bootloader is responsible for processor- and board-specific initializations, loading a kernel and an optional initial ramdisk into memory and passing control to the kernel. In addition, a bootloader might be in charge of providing BIOS services, performing POST, supporting firmware downloads to the target, and passing memory layout and configuration information to the kernel. On embedded devices that use encrypted firmware images for security reasons, bootloaders may have the task of decrypting firmware. Some bootloaders support a debug monitor to load and debug stand-alone code on to the target device. You may also decide to build a failure-recovery mechanism into your bootloader to recoup from kernel corruption on the field.

In general, bootloader architecture depends on the processor family, the chipsets present on the hardware platform, the boot device, and the operating system running on the device. To illustrate the effects of the processor family on the boot suite, consider the following:

- A bootloader for a device designed around the StrongARM processor has to know whether it's booting the system or waking it up from sleep, because the processor starts execution from the top of its address space (the bootloader) in both cases. The bootloader has to pass control to the kernel code that restores the system state if it's waking up from sleep or load the kernel from the boot device if the system is starting from reset.
- An x86 bootloader might need to switch to protected mode to load a kernel bigger than the 1MB real-mode limit.
- Embedded systems not based on x86 platforms cannot avail the services of a legacy BIOS. So, if you want your embedded device to boot, for example, from an external USB device, you have to build USB

capabilities into your bootloader.

- Even when two platforms are based on similar processor cores, the bootloader architecture may differ based on the SoC. For example, consider two ARM-based devices, the Compaq iPAQ H3900 PDA and the Darwin Jukebox. The former is built around the Intel PXA250 controller chip, which has an XScale CPU based on an ARMv5 core, and the latter is designed using the Cirrus Logic EP7312 controller that uses an ARMv3 core. Whereas XScale supports JTAG (named after the *Joint Test Action Group*, which developed this hardware-assisted debugging standard) to load a bootloader onto flash, the EP7312 has a bootstrap mode to accomplish the same task.

The boot suite needs a mechanism to transfer a bootloader image from the host development system to the target's boot device. This is called *bootstrapping.* Bootstrapping is straightforward on PC-compatible systems where the BIOS flash is programmed using an external burner if it's corrupted or updated after booting into an operating system if it's healthy. Embedded devices, however, do not have a generic method for bootstrapping.

To illustrate bootstrapping on an embedded system, take the example of the Cirrus Logic EP7211 controller (which is the predecessor of the EP7312 discussed in the previous section). The EP7211 executes code from a small internal 128-byte memory when it's powered on in a bootstrap mode. This 128-byte code downloads a bootstrap image from a host via the serial port to an on-board 2KB SRAM and transfers control to it. The boot suite has to be thus architected into three stages, each loaded at a different address:

- The first stage (the 128-byte image) is part of processor firmware.
- The second stage lives in the on-chip SRAM, so it can be up to 2KB. This is the bootstrapper.
- The bootstrapper downloads the actual bootloader image from an external host to the top of flash memory. The bootloader gets control when the processor powers on in normal operation mode.

Note that the processor-resident microcode (the first stage) itself cannot function as the bootstrapper because a bootstrapper needs to have the capability to program flash memory. Because many types of flash chips can be used with a processor, the bootstrapper code needs to be board-specific.

Many controller chips do not support a bootstrap mode. Instead, the bootloader is written to flash via a JTAG interface. You can use your JTAG debugger's command interface to access the processor's debug logic and burn the bootloader to the target device's flash memory. We will have a more detailed discussion on JTAG debugging in the section "JTAG Debuggers" in Chapter 21, "Debugging Device Drivers."

There are controllers that support both bootstrap execution mode and JTAG. The Freescale i.MX21 (and its upgraded version i.MX27) based on an ARM9 core is one such controller.

After a bootloader is resident on flash, it can update itself as well as other firmware components such as the kernel and the root filesystem. The bootloader can directly talk to a host machine and download firmware components via interfaces such as UART, USB, or Ethernet.

Table 18.1 looks at a few example Linux bootloaders for ARM, PowerPC, and x86.

Table 18.1. Linux Bootloaders

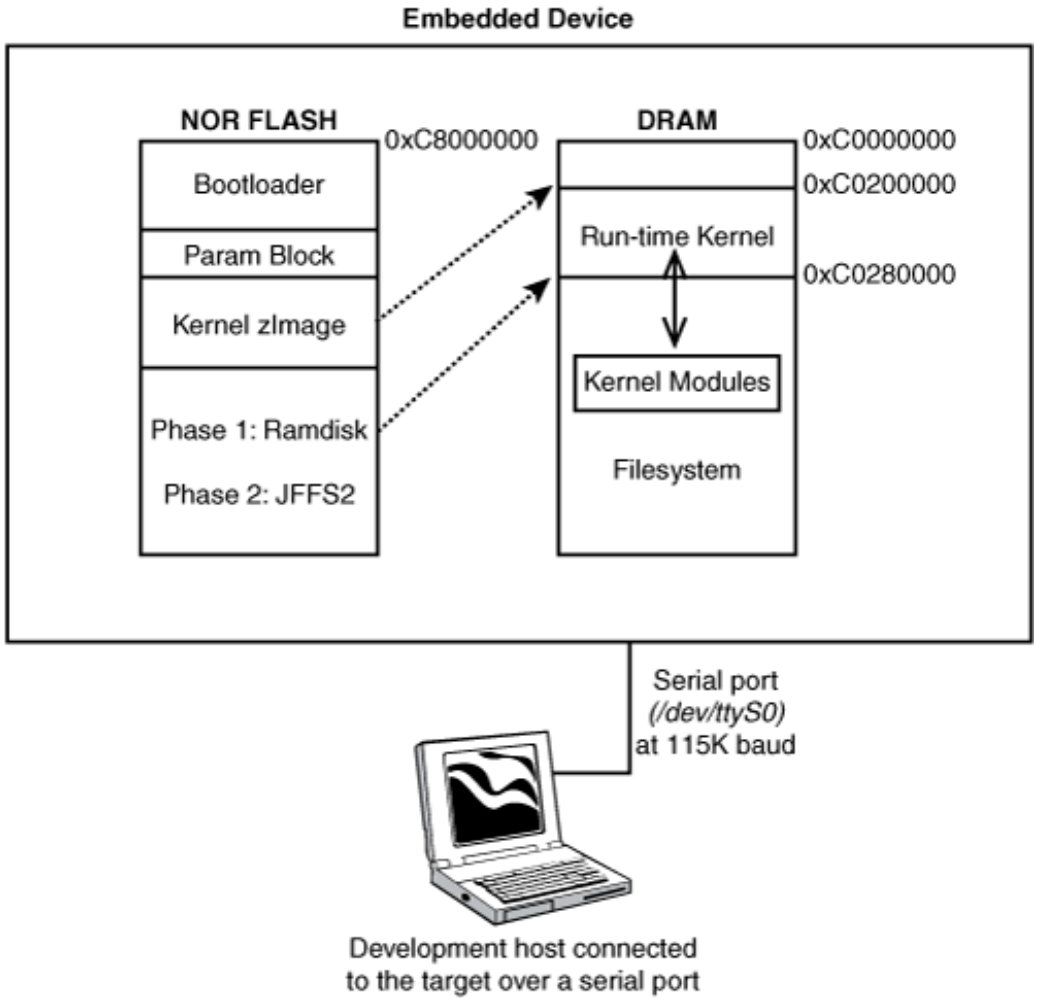
Processor Platform	Linux Bootloaders
ARM	RedBoot (www.cygwin.com/redboot) is a bootloader popular on ARM-based hardware. Redboot is based on a hardware abstraction offered by the eCos operating system (http://ecos.sourceware.org/). The *BootLoader Object* or BLOB (http://sourceforge.net/projects/blob/), a bootloader originally developed for StrongARM-based boards, is commonly custom ported to other ARM-based platforms, too. BLOB is built as two images, one that performs minimal initializations, and the second that forms the bulk of the bootloader. The first image relocates the second to RAM, so the bootloader can easily upgrade itself.
PowerPC	PowerPC chips used on embedded devices include SoCs such as IBM's 405LP and the 440GP, and Motorola's MPC7xx and MPC8xx. Bootloaders such as U-Boot (http://sourceforge.net/projects/u-boot/), SLOF, and PIBS boot Linux on PowerPC-based hardware.
x86	Most x86-based systems boot from disk drives. Embedded x86 boards may boot from solid-state disks rather than mechanical drives. The first stage of a disk-resident bootloader consists of a sector-sized chunk that is loaded by the BIOS. This is called the *Master Boot Record* (MBR) and contains up to 446 bytes of code, four partition table entries consuming 16 bytes each, and a 2-byte signature (thus making up a 512-byte sector). The MBR is responsible for loading the second stage of the bootloader. Each intervening stage has its own tasks, but the final stage lets you choose the kernel image and command-line arguments, loads the kernel and any initial ramdisk to memory, and transfers control to the kernel. As an illustration, let's look at three bootloaders popularly used to boot Linux on x86-based hardware: • The *Linux Loader* or LILO (http://freshmeat.net/projects/lilo/) is packaged along with some Linux distributions. When the first stage of the bootloader is written to the boot sector, LILO precalculates the disk locations of the second stage and the kernel. If you build a new kernel image, you have to rewrite the boot sector. The second stage allows the user to interactively select the kernel image and configure command-line arguments. It then loads the kernel to memory. • GRUB (www.gnu.org/software/grub) is different from LILO in that the kernel image can live in any supported filesystem, and the boot sector need not be rewritten if the kernel image changes. GRUB has an extra stage 1.5 that understands the filesystem holding the boot images. Currently supported filesystems are EXT2, DOS FAT, BSD FFS, IBM JFS, SGI XFS, Minix, and Reiserfs. GRUB complies with the Multiboot specification, which allows any complying operating system to boot via any complying bootloader. You looked at a sample GRUB configuration file in Chapter 2, "A Peek Inside the Kernel." • SYSLINUX (http://syslinux.zytor.com/) is a no-frills Linux bootloader. It understands the FAT filesystem, so you can store the kernel image and the second stage bootloader on a FAT partition.

Giving due thought to the design and architecture of the bootloader suite lays a solid foundation for embedded software development. The key is to choose the right bootloader as your starting point. The benefits range from a shorter software development cycle to a feature-rich and robust device.

Memory Layout

Figure 18.2 shows an example memory layout on an embedded device. The bootloader sits on top of the NOR flash. Following the bootloader lies the *param* block, a statically compiled binary image of kernel command-line arguments. The compressed kernel image comes next. The filesystem occupies the rest of the available flash memory. In the initial phase, when you start development with a first-shot kernel, the filesystem is usually a compressed ramdisk (*initrd* or *initramfs*), because having a flash-based filesystem entails getting the kernel MTD subsystem configured and running.

Figure 18.2. Example memory layout on an embedded device.



During power-on, the bootloader in Figure 18.2 uncompresses the kernel and loads it to DRAM at `0xc0200000`. It then loads the ramdisk at `0xc0280000` (unless you build an *initramfs* into the base kernel as you learned in Chapter 2). Finally, it obtains command-line arguments from the *param* block and transfers control to the kernel.

Because you may have to work with unconventional consoles and memory partitions on embedded devices, you have to pass the right command-line arguments to the kernel. For the device in Figure 18.2, this is a possible command line:

```
console=/dev/ttyS0,115200n8 root=/dev/ram initrd=0xC0280000
```

When you have the kernel MTD drivers recognizing your flash partitions, the area of flash that holds the ramdisk can instead contain a JFFS2-based filesystem. With this, you don't have to load the initrd to DRAM. Assuming that you have mapped the bootloader, param block, kernel, and filesystem to separate MTD partitions, the command line now looks like this:

```
console=/dev/ttyS0,115200n8 root=/dev/mtdblock3
```

See the sidebar "ATAGs" for another method of passing parameters from the bootloader to the kernel.

ATAGs

On ARM kernels, command-line arguments are deprecated in favor of a tagged list of parameters. This mechanism, called ATAG, is described in *Documentation/arm/Booting*. To pass a parameter to the kernel, create the corresponding tag in system memory from the bootloader, supply a kernel function to parse it, and add the latter to the list of tag parsing functions using the `__tagtable()` macro. The `tag` structure and its relatives are defined in *include/asm-arm/setup.h*, whereas *arch/arm/kernel/setup.c* contains functions that parse several predefined ATAGs.

Kernel Porting

Like setting up tool chains, porting the kernel to your target device was a serious affair a few years ago. One had to evaluate the stability of the current kernel tree for the architecture of interest, apply available patches that were not yet part of the mainline, make modifications, and hope for good luck. But today, you are likely to find a close starting point, not just for your SoC, but for a hardware board that is similar to yours. For example, if you are designing an embedded device around the Freescale i.MX21 processor, you have the option of starting off with the kernel port (*arch/arm/-mach-imx/*) for the i.MX21-based reference board built by the processor vendor. If you thus start development from a suitable distribution-supplied or standard kernel available for a board that resembles yours, chances are, you won't have to grapple with complex kernel bring-up issues.

But even with a close match, you are likely to face issues caused by modified memory maps, changed chip selects, board-specific GPIO assignments, dissimilar clock sources, disparate flash banks, timing requirements of a new LCD panel, or a different debug UART port. A change in clocking for example, can ripple through dozens of registers and impact the operation of several I/O peripherals. You might need an in-depth reading of the CPU reference manual to resolve it. To figure out a modified interrupt pin routing caused by a different GPIO assignment, you might have to pore over your board schematics. To program an LCD controller with `HSYNC` and `VSYNC` durations appropriate to your LCD panel, you may need to connect an oscilloscope to your board and digest the information that it gathers.

Depending on the demands on your device, you may also need to make kernel changes unrelated to bring up. It could be as simple as exporting some information via procfs or as complex as modifying the kernel for fast boot.

After you have the base kernel running, you can turn your attention to enabling device drivers for the different I/O interfaces on your hardware.

uClinux

uClinux is a branch of the Linux kernel intended for lower-end microprocessors that have no Memory Management Units (MMUs). uClinux ports are available for processors such as H8, Blackfin, and Dragonball. Most portions of uClinux are merged with the mainline 2.6 kernel.

The uClinux project is hosted at www.uclinux.org. The website contains patches, documentation, the code repository, list of supported architectures, and information for subscribing to the uclinux-dev mailing list.

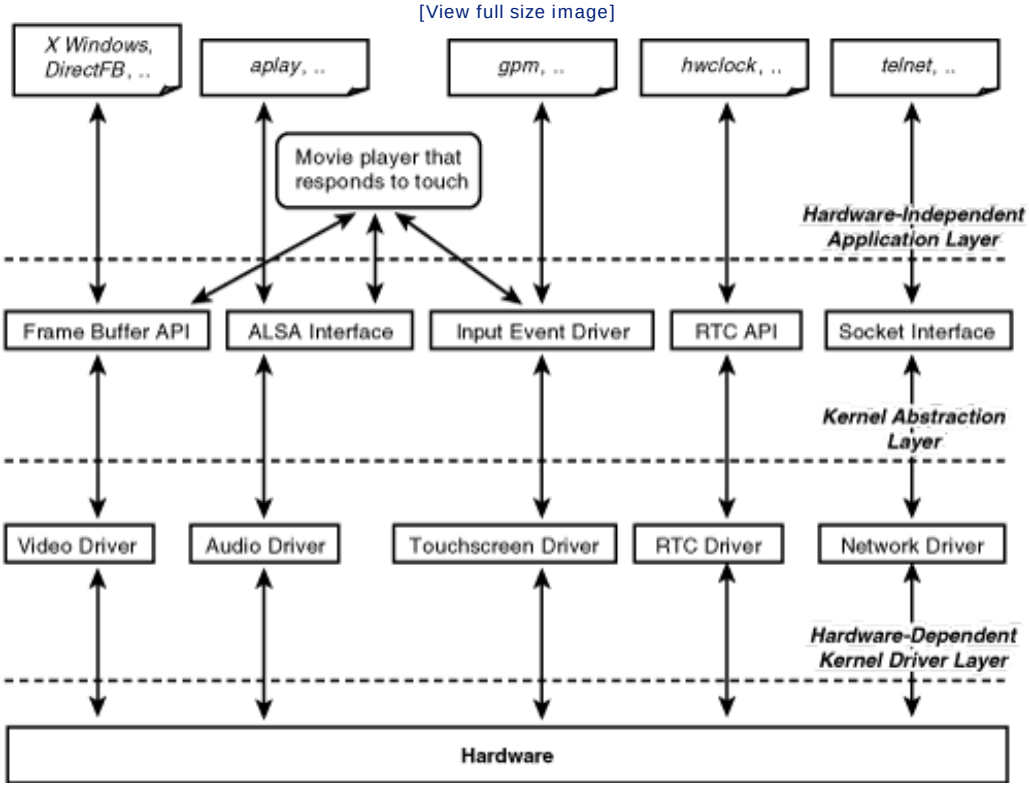
Embedded Drivers

One of the reasons Linux is so popular in the embedded space is that its formidable application suite works regardless of the hardware platform, thanks to kernel abstraction layers that lie beneath them. So, as shown in Figure 18.3, all you need to do to get a feature-rich embedded system is to implement the low-level device drivers ensconced between the abstraction layers and the hardware. You need to do one of the following for each peripheral interface on your device:

- Qualify an existing driver. Test and verify that it works as it's supposed to.
- Find a driver that is a close match and modify it for your hardware.
- Write a driver from scratch.

Figure 18.3. Hardware-independent applications and hardware-dependent drivers.

[View full size image]



Assuming a kernel engineer participates in component selection, you're likely to have existing drivers or close enough matches for most peripheral devices. To take advantage of existing drivers, go through the block diagram and schematics of your hardware, identify the different chipsets, and cobble together a working kernel configuration file that enables the right drivers. Based on your footprint or boot time requirements, modularize possible device drivers or build them into the base kernel.

To learn about device drivers for I/O interfaces commonly found on embedded hardware, let's take a clockwise tour around the embedded controller shown in Figure 18.1, starting with the NOR flash.

Flash Memory

Embedded devices such as the one in Figure 18.2, boot from flash memory and have filesystem data resident on flash-based storage. Many devices use a small NOR flash component for the former and a NAND flash part for the latter.[2] NOR memory, thus, holds the bootloader and the base kernel, whereas NAND storage contains filesystem partitions and device driver modules.

> [2] In today's embedded market where the Bill Of Material (BOM) cost is often all-important, it's not uncommon for devices to contain only NAND storage. Such devices boot from NAND flash and have their filesystems also reside in NAND memory. NAND boot needs support from both the processor and the bootloader.

Flash drivers are supported by the kernel's MTD subsystem discussed in Chapter 17, "Memory Technology Devices." If you're using an MTD-supported chip, you need to write only an MTD map driver to suitably partition the flash to hold the bootloader, kernel, and filesystem. Listings 17.1, 17.2, and 17.3 in Chapter 17 implement a map driver for the Linux handheld, as shown in Figure 17.2 of the same chapter.

UART

The UART is responsible for serial communication and is an interface you are likely to find on all microcontrollers. UARTs are considered basic hardware, so the kernel contains UART drivers for all microcontrollers on which it runs. On embedded devices, UARTs are used to interface the processor with debug serial ports, modems, touch controllers, GPRS chipsets, Bluetooth chipsets, GPS devices, telemetry electronics, and so on.

Look at Chapter 6, "Serial Drivers," for a detailed discussion on the Linux serial subsystem.

Buttons and Wheels

Your device may have several miscellaneous peripherals such as keypads (micro keyboards organized in the common QWERTY layout, data-entry devices having overloaded keys as found in cell phones, keypads having ABC-type layout, and so on), LEDs, roller wheels, and buttons. These I/O devices interface with the CPU via GPIO lines or a CPLD (see the following "CPLD/FPGA" section). Drivers for such peripherals are usually straightforward char or misc drivers. Some of the drivers export device-access via procfs or sysfs rather than through */dev* nodes.

PCMCIA/CF

A PCMCIA or CF slot is a common add-on to embedded devices. The advantage of, say, WiFi enabling an embedded device using a CF card is that you won't have to respin the board if the WiFi controller goes end of life. Also, because diverse technologies are available in the PCMCIA/CF form factor, you have the freedom to change the connectivity mode from WiFi to another technology such as Bluetooth later. The disadvantage of such a scheme is that even with mechanical retaining, sockets are inherently unreliable. There is the possibility of the card coming loose due to shock and vibe, and resulting intermittent connections.

PCMCIA and CF device drivers are discussed in Chapter 9, "PCMCIA and Compact Flash."

SD/MMC

Many embedded processors include controllers that communicate with SD/MMC media. SD/MMC storage is built using NAND flash memory. Like CF cards, SD/MMC cards add several gigabytes of memory to your device. They

also offer an easy memory upgrade path, because the available density of SD/MMC cards is constantly increasing.

Chapter 14, "Block Drivers," points you to the SD/MMC subsystem in the kernel.

USB

Legacy computers support the USB host mode, by which you can communicate with most classes of USB devices. Embedded systems frequently also require support for the USB device mode, wherein the system itself functions as a USB device and plugs into other host computers.

As you saw in Chapter 11, "Universal Serial Bus," many embedded controllers support USB OTG that lets your device work either as a USB host or as a USB device. It allows you, for example, to connect a USB pen drive to your embedded device. It also allows your embedded device to function as a USB pen drive by exporting part of its local storage for external access. The Linux USB subsystem offers drivers for USB OTG. For hardware that is not compatible with OTG, the USB Gadget project, now part of the mainline kernel, brings USB device capability.

RTC

Many embedded SoCs include RTC support to keep track of wall time, but some rely on an external RTC chip. Unlike x86-based computers where the RTC is part of the South Bridge chipset, embedded controllers commonly interface with external RTCs via slow serial buses such as I^2C or SPI. You can drive such RTCs by writing client drivers that use the services of the I^2C or SPI core as discussed in Chapter 8, "The Inter-Integrated Circuit Protocol." Chapter 2 and Chapter 5, "Character Drivers," discussed RTC support on x86-based systems.

Audio

As you saw in Chapter 13, "Audio Drivers," an audio codec converts digital audio data to analog sound signals for playback via speakers and performs the reverse operation for recording through a microphone. The codec's connection with the CPU depends on the digital audio interface supported by the embedded controller. The usual way to communicate with a codec is via buses, such as AC'97 or I^2S.

Touch Screen

Touch is the primary input mechanism on several embedded devices. Many PDAs offer soft keyboards for data entry. In Chapter 6, we developed a driver for a serial touch controller, and in Chapter 7, "Input Drivers," we looked at a touch controller that interfaced with the CPU via the SPI bus.

If your driver conforms to the *input* API, it should be straightforward to tie it with a graphical user interface. You might, however, need to add custom support to calibrate and linearize the touch panel.

Video

Some embedded systems are headless, but many have associated displays. A suitably oriented (landscape or portrait) LCD panel is connected to the video controller that is part of the embedded SoC. Many LCD panels come with integrated touch screens.

As you learned in Chapter 12, "Video Drivers," frame buffers insulate applications from display hardware, so porting a compliant GUI to your device is easy, as long as your display driver conforms to the frame buffer interface.

CPLD/FPGA

Complex Programmable Logic Devices (CPLDs) or their heavy-duty counterparts, *Field Programmable Gate Arrays* (FPGAs), can add a thick layer of fast OS-independent logic. You can program CPLDs (and FPGAs) in a language such as *Very high speed integrated circuit Hardware Description Language* (VHDL). Electrical signals between the processor and peripherals propagate through the CPLD, so by appropriately programming the CPLD, the OS obtains elegant register interfaces for performing complex I/O. The VHDL code in the CPLD internally latches these register contents onto the data bus after performing necessary control logic.

Consider, for example, an external serial LCD controller that has to be driven by shifting in each pixel bit. The Linux driver for this device will have a tough time toggling the clock and wiggling I/O pins several times for sending each pixel or command byte to the serial LCD controller. If this LCD controller is routed to the processor via a CPLD, however, the VHDL code can perform the necessary serial shifting by clocking each bit in and present a parallel register interface to the OS for command and data. With these virtual LCD command and data registers, the LCD driver implementation is rendered simple. Essentially, the CPLD converts the cumbersome serial LCD controller to a convenient, parallel one.

If the CPLD engineer and the Linux driver developer collaborate, they can arrive at an optimum partitioning between the VHDL code and the Linux driver that'll save time and cost.

Connectivity

Connectivity injects intelligence, so there are few embedded devices that have no communication capability. Popular networking technologies found on embedded devices include WiFi, Bluetooth, cellular modems, Ethernet, and radio communication.

Chapter 15, "Network Interface Cards," explored device drivers for wired networking, and Chapter 16, "Linux Without Wires," looked at drivers for wireless communication technologies.

Domain-Specific Electronics

Your device is likely to contain electronics specific to the usage industry domain. It could be a telemetry interface for a hospital-grade device, a sensor for automotive hardware, biometrics for a security gadget, GPRS for a cellular phone, or GPS for a navigation system. These peripherals usually communicate with the embedded controller over standard I/O interfaces such as UART, USB, I^2C, SPI, or *controller area network* (CAN). For devices interfacing via a UART, you often have little work to do at the device driver level because the UART driver takes care of the communication. For devices such as a fingerprint sensor that interface via USB, you may have to write a USB client driver. You might also face proprietary interfaces, such as a switching fabric for a network processor, in which case, you may need to write a full-fledged device driver.

Consider the digital media space. Cable or *Direct-to-home* (DTH) interface systems are usually built around *set-top box* (STB) chipsets. These chips have capabilities such as personal video recording (recording multiple channels to a hard disk, recording a channel while viewing another and so forth) and conditional access (allowing the service provider to control what the end user sees depending on subscription). To achieve this, STB chips have a processor core coupled with a powerful graphics engine. The latter implements MPEG codecs in hardware. Such audio-video codecs can decode compressed digital media standards such as MPEG2 and MPEG4. (MPEG is an acronym for *Moving Picture Experts Group*, the body responsible for developing motion picture standards.) If you are embedding Linux onto an STB, you will need to drive such audio-video codecs.

More Drivers

If your device serves a life-critical industry domain such as health care, the system memory might have ECC capabilities. Chapter 20, "More Devices and Drivers," discusses ECC reporting.

If your embedded device is battery powered, you may want to use a suitable CPU frequencygovernor to dynamically scale processor frequency and save power. Chap-ter 20 also discusses CPU frequency drivers and power management.

Most embedded processors have a built-in hardware watchdog that recovers the system from freezes. You

looked at watchdog drivers in Chapter 5. Use a suitable driver from *drivers/char/watchdog/* as the starting point to implement a driver for your system's watchdog.

If your embedded device contains circuitry to detect *brownout*,[3] you might need to add capability to the kernel to sense that condition and take appropriate action.

[3] *Brownout* is the scenario when input voltage drops below tolerable levels. (*Blackout*, on the other hand, refers to total loss of power.) Brownout detection is especially relevant if your device is powered by a technology such as Power over Ethernet (PoE) rather than a conventional wall socket.

Several embedded SoCs contain built-in *pulse-width modulator* (PWM) units. PWMs let you digitally control analog devices such as buzzers. The voltage level supplied to the target device is varied by programming the PWM's duty cycle (the On time of the PWM's output waveform relative to its period). LCD brightness is another example of a feature controllable using PWMs. Depending on the target device and the usage scenario, you can implement char or misc driver interfaces to PWMs.

The Root Filesystem

Before the advent of Linux distributions, it used to be a project by itself to put together a compact application-set tailored to suit the size limitations of available storage. One had to cobble together the sources of a minimal set of utilities, libraries, tools, and daemons; ensure that their versions liked each other; and cross-compile them. Today's distributions supply a ready-made application-set built for supported processors and offer tools that let you pick and choose components at the granularity of packages. Of course, you may still want to implement custom utilities and tools to supplement the distribution-supplied applications.

On embedded devices, flash memory (discussed in Chapter 17) is the commonly used vehicle to hold the application-set and is mounted as the root device at the end of the boot process. Hard disks are uncommon because they are power-intensive, bulky, and have moving parts that are not tolerant to shock and vibe. Common places that hold the root filesystem on embedded devices include the following:

- An initial ramdisk (*initramfs* or *initrd*) is usually the starting point before you get drivers for other potential root devices working and is used for development purposes.
- NFS-mounting the root filesystem is a development strategy much more powerful than using a ramdisk. We discuss this in detail in the next section.
- Storage media such as flash chips, SD/MMC cards, CF cards, DOCs, and DOMs.

Note that it may not be a good idea to let all the data stay in the root partition. It's common to spread files across different storage partitions and tag desired *read-write* or *read-only* protection flags, especially if there is the possibility that the device will be shut down abruptly.

NFS-Mounted Root

NFS-mounting the root filesystem can serve as a catalyst to hasten the embedded development cycle. In this case, the root filesystem physically resides on your development host and not on the target, so its size is virtually unlimited and not restricted by the amount of storage locally available on the target. Downloading device driver modules or applications to the target, as well as uploading logs, is as simple (and fast) as copying them to */path/to/target/rootfilesystem/* on your development host. Such ease of testing and debugging is a good reason why you should insist on having Ethernet on engineering-level hardware, even if production units won't have Ethernet support. Having Ethernet on your board also lets your bootloader use the Trivial File Transfer Protocol (TFTP) to download the kernel image to the target over a network.

Table 18.2[4] shows the typical steps needed to get TFTP and NFS working with your embedded device. It assumes that your development host also doubles up as TFTP, NFS, and DHCP servers, and that the bootloader (BLOB in this example) supports the Ethernet chipset used on the embedded device.

[4] The filenames and directory path names used in Table 18.2 are distribution-dependent.

Table 18.2. Saving Development Time with TFTP and NFS

	Target Embedded Device	Host Development Platform

Kernel Boot over TFTP	Configure the IP address of the target and the server (host) from the bootloader prompt:
	`/* Target IP */`
	`blob> ip 4.1.1.2`
	`/* Host IP */`
	`blob> server 4.1.1.1`
	`/* Kernel image */`
	`blob> TftpFile /tftpdir/zImage`
	`/* Pull the Kernel over the net */`
	`blob> tftp`
	`TFTPing /tftpboot/zImage.....Ok`
	`blob>`
		Configure the host IP address:
		`bash> ifconfig eth0 4.1.1.1`
		
		Install and configure the TFTP server (the exact steps depend on your distribution):
		
		`bash> cat /etc/xinetd.conf/tftp`
		`service tftp`
		`{`
		`  socket_type  = dgram`
		`  protocol     = udp`
		`  wait         = yes`
		`  user         = root`
		`  server       = /usr/sbin/in.tftpd`
		`  server_args  = /tftpdir`
		`  disable      = no`
		`  per_source   = 11`
		`  cps          = 100 2`
		`  flags        = IPv4`
		`}`
		
		Make sure that the TFTP server is present in */usr/sbin/in.tftpd* and that *xinetd* is alive.
		
		Compile the target kernel with NFS enabled and copy it to */tftpdir/zImage*.

Root filesystem over NFS	**`blob> boot console=/dev/ttyS0,115200n8 root=/dev/nfs ip=dhcp`**
	`/*Kernel boot messages*/`
	`/* ... */`
	`VFS: Mounted root (nfs filesystem)`
	`/* ... */`
	`login:`
		Export */path/to/target/root/* for NFS access:
		`bash> cat /etc/exports`
		`/path/to/target/root/ *(rw,sync,no_root_squash,no_all_squash)`
		
		Start NFS:
		
		`bash> service nfs start`
		
		Configure the DHCP server. The kernel on the embedded device relies on this server to assign it the 4.1.1.2 IP address during boot and to supply */path/to/target/root/:*
		Code View:
		`bash> cat /etc/dhcpd.conf`
		`...`
		`subnet 4.1.1.0 netmask 255.255.255.0 {`
		`range 4.1.1.2 4.1.1.10`
		`max-lease-time 43200`
		`option routers 4.1.1.1`
		`option ip-forwarding off`

```
option broadcast-address 4.1.1.255
option subnet-mask 255.255.255.0
group {
  next-server 4.1.1.1
  host target-device {
   /* MAC of the embedded device */
   hardware Ethernet AA:BB:CC:DD:
   EE:FF;
   fixed-address 4.1.1.2;
   option root-path
   "/path/to/target/root/";
  }
}
...
bash> service dhcpd start
bash>
```

Compact Middleware

Embedded devices that are tight on memory prefer middleware implementations that have small footprint and low runtime memory requirements. The trade-offs usually lie in features, standards compatibility, and speed. Let's take a look at some popular compact middleware solutions that may be potential candidates for populating your root filesystem.

BusyBox is a tool commonly used to provide a multi-utility environment on embedded systems having limited memory. It scratches out some features but provides an optimized replacement for several shell utilities.

uClibc is a compact version of the GNU C library that was originally developed to work with uClinux. uClibc works on normal Linux systems, too, and is licensed under LGPL. If your embedded device is short on space, try uClibc rather than glibc.

Embedded systems that need to run an X Windows server commonly rely on *TinyX*, a low-footprint X server shipped along with the XFree86 4.0 code. TinyX runs over frame buffer drivers and can be used on devices, such as the one showed in Figure 12.6 of Chapter 12.

Thttpd is a lightweight HTTP server that makes low demands on CPU and memory resources.

Even if you are creating a non-Linux solution using a tiny 8-bit MMU-less microcontroller, you will likely want it to interoperate with Linux. Assume, for example, that you are writing deeply embedded firmware for an Infrared storage keychain. The keychain can hold a gigabyte of personal data that can be accessed via a web browser from your Linux laptop over Infrared. If you are running a compact TCP/IP stack, such as *uIP* over a minimal IrDA stack such as *Pico-IrDA* on the Infrared keychain, you have the task of ensuring their interoperability with the corresponding Linux protocol stacks.

Table 18.3 lists the home pages of the compact middleware projects referred to in this section.

Table 18.3. Examples of Compact Middleware

Name	Description	Download Location

Name	Description	Download Location
BusyBox	Small footprint shell environment	www.busybox.net
uClibc	Small-sized version of glibc	www.uclibc.org
TinyX	X server for devices that are tight on memory	Part of the X Windows source tree downloadable from ftp://ftp.xfree86.org/pub/XFree86/4.0/
Thttpd	Tiny HTTP server	www.acme.com/software/thttpd
uIP	Compact TCP/IP stack for microcontrollers	www.sics.se/~adam/uip
Pico-IrDA	Minimal IrDA stack for microcontrollers	http://blaulogic.com/pico_irda.shtml

Test Infrastructure

Most industry domains that use embedded devices are governed by regulatory bodies. Having an extensible and robust test infrastructure is likely to be as important as implementing modifications to the kernel and device drivers. Broadly, the test framework is responsible for the following:

1. Demonstrating compliance to obtain regulatory approvals. If your system is a medical-grade device for the U.S. market, for example, you should orient your test suite for getting approvals from the *Food and Drug Administration* (FDA).

2. Most electronic devices intended for the U.S. market have to comply with emission standards such as *electromagnetic interference* (EMI) and *electromagnetic compatibility* (EMC) as laid down by the *Federal Communications Commission* (FCC). To demonstrate compliance, you may need to run a battery of tests inside a chamber that models different operating environments. You might also have to verify that the system runs normally when an electrostatic gun is pointed at different parts of the board.

3. Build verification tests. Whenever you build a software deliverable, subject it to *quality assurance* (QA) using these tests.

4. Manufacturing tests. Each time a device is assembled, you have to verify its functionality using a set of tests. These tests assume significance when manufacturing moves into volume production.

To have a common test base for all these, it's a good idea to implement your test harness over Linux, rather than develop it as a stand-alone suite. Stand-alone code is not easily scalable or extendable. Adding a simple test to ping the next-hop router is a five-line script on a Linux-based test system but can entail writing a network driver and a protocol stack if you are using a stand-alone test monitor.

A test engineer need not be a kernel guru but will need to imbibe implementation information from the development team and think critically.

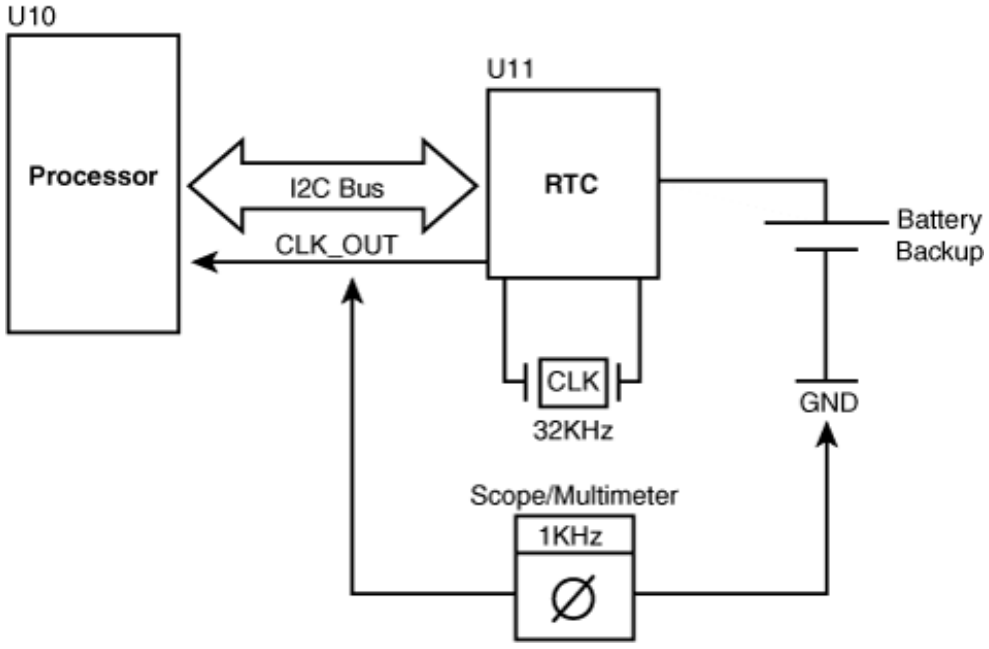
Debugging

Before closing this chapter, let's visit a few topics related to debugging embedded software.

Board Rework

Navigating board schematics and datasheets is an important debugging skill you need while bringing up the bootloader or kernel on embedded hardware. Understanding your board's *placement plot*, which is a file that shows the position of chips on your board, is a big help when you are debugging a potential hardware problem using an oscilloscope, or when you need to perform minor board rework. *Reference designators* (such as U10 and U11 in Figure 18.4) associate each chip in the schematic with the placement plot. *Printed circuit boards* (PCBs) are usually clothed with *silk screens* that print the reference designator near each chip.

Figure 18.4. Debugging an I^{2}C RTC on an embedded system.



Consider this fictitious scenario where USB enumeration doesn't occur on your board under test. The USB hub driver detects device insertions but is not able to assign endpoint addresses. A close look at the schematics reveals that the connections originating from the `SPEED` and `MODE` pins of the USB transceiver have been interchanged by mistake. An examination of the placement plot identifies the location of the transceiver on the PCB. Matching the transceiver's reference designator on the placement plot with the silk screen on the PCB pinpoints the places where you have to solder "yellow wires" to repair the faulty connections.

A multimeter and an oscilloscope are worthy additions to your embedded debugging toolkit. As an illustration, let's consider an example situation involving the I^{2}C RTC, as shown in Figure 8.3 of Chapter 8. That figure is reproduced there with a multimeter/scope attached to probe points of interest. Consider this scenario: You have written an I^{2}C client driver for this RTC chip as described in the section "Device Example: Real Time Clock" in Chapter 8. However, when you run your driver on the board, it renders the system unbootable. Neither does the bootloader come up when you reset the board, nor does your JTAG debugger connect to the target. To

understand possible causes of this seemingly fatal error, let's take a closer look at the connection diagram. Because both the RTC and the CPU need an external clock, the board supplies it using a single 32KHz crystal. This 32KHz clock needs to be buffered, however. The RTC buffers the clock for its use and makes it available on an output pin for free. This pin, `CLK_OUT`, feeds the clock to the processor. Connect an oscilloscope (or a multimeter that can measure frequency) between `CLK_OUT` and ground to verify the processor clock frequency. As you can see in Figure 18.4, the scope reads 1KHz rather than the expected 32KHz! What could be wrong here?

The RTC control register contains bits that choose the frequency of `CLK_OUT`. While probing the chip (on the lines of `myrtc_attach()` in Chapter 8), the driver has erroneously initialized these bits to generate 1KHz on `CLK_OUT`. RTC registers are nonvolatile because of the battery backup, so the control register holds this bad value across reboots. The resulting skewed clock is sufficient to render the system unbootable. Disconnect the RTC's backup battery, drain the registers, reconnect the battery, verify using the scope that the 32KHz clock is restored on `CLK_OUT`, fix your driver code, and start afresh!

Debuggers

You can use most of the debugging techniques that you will learn in Chapter 21 while embedding Linux. Kernel debuggers are available for several processor platforms. JTAG debuggers, also explored in Chapter 21, are more powerful than kernel debuggers and are popularly used in the embedded space to debug the bootloader, base kernel, and device-driver modules.

Chapter 19. Drivers in User Space

In This Chapter

- Process Scheduling and Response Times 553
- Accessing I/O Regions 558
- Accessing Memory Regions 562
- User Mode SCSI 565
- User Mode USB 567
- User Mode I^2C 571
- UIO 573
- Looking at the Sources 574

Most device drivers prefer to lead a privileged life inside the kernel, but some are at home in the indeterministic world outside. Several kernel subsystems, such as SCSI, USB, and I^2C, offer some level of support for user mode drivers, so you might be able to control those devices without writing a single line of kernel code.

In spite of the inclement weather in user land, user mode drivers enjoy certain advantages. They are easy to develop and debug. You won't have to reboot the system every time you dereference a dangling pointer. Some user mode drivers will even work across operating systems if the device subsystem enjoys the services of a standard user-space programming library. Here are some rules of thumb to help decide whether your driver should reside in user space:

- Apply the possibility test. What can be done in user space should probably stay in user space.
- If you have to talk to a large number of slow devices and if performance requirements are modest, explore the possibility of implementing the drivers in user space. If you have time-critical performance requirements, stay inside the kernel.
- If your code needs the services of kernel APIs, access to kernel variables, or is intertwined with interrupt handling, it has a strong case for being in kernel space.
- If much of what your code does can be construed as policy, user land might be its logical residence.
- If the rest of the kernel needs to invoke your code's services, it's a candidate for staying inside the kernel.
- You can't easily do floating-point arithmetic inside the kernel. *Floating-point unit* (FPU) instructions can, however, be used from user space.

That said, you can't accomplish too much from user space. Many important device classes, such as storage media and network adapters, cannot be driven from user land. But even when a kernel driver is the appropriate solution, it's a good idea to model and test as much code as you can in user space before moving it to kernel space. The testing cycle is faster, and it's easier to traverse all possible code paths and ensure that they are clean.

In this chapter, the term *user space driver* (or *user mode driver*) is used in a generic sense that does not strictly conform to the semantics of a driver implied thus far in the book. An application is considered to be a user mode driver if it's a candidate for being implemented inside the kernel, too.

The 2.6 kernel overhauled a subsystem that is of special interest to user space drivers. The new process scheduler offers huge response-time benefits to user mode code, so let's start with that.

Process Scheduling and Response Times

Many user mode drivers need to perform some work in a time-bound manner. In user space, indeterminism due

to scheduling and paging often come in the way of fast response times, however. To see how you can minimize the impact of the former hurdle, let's dip into recent Linux schedulers and understand their underlying philosophy.

The Original Scheduler

In the 2.4 and earlier days, the scheduler used to recalculate scheduling parameters of each task before taking its pick. The time consumed by the algorithm thus increased linearly with the number of contending tasks in the system. In other words, it used `O(n)` time, where `n` is the number of active tasks. On a system running at high loads, this translated to significant overhead. The 2.4 algorithm also didn't work very well on SMP systems.

The O(1) Scheduler

Time consumed by an `O(n)` algorithm depends linearly on the size of its input, and an $O(n^2)$ solution depends quadratically on the length of its input, but an `O(1)` technique is independent of the input and thus scales well. The 2.6 scheduler replaced the `O(n)` algorithm with an `O(1)` method. In addition to being super-scalable, the scheduler has built-in heuristics to improve user responsiveness by providing preferential treatment to tasks involved in I/O activity. Processes are of two kinds: I/O bound and CPU bound. I/O-bound tasks are often sleep-waiting for device I/O, while CPU-bound ones are workaholics addicted to the processor. Paradoxically, to achieve fast response times, lazy tasks get incentives from the `O(1)` scheduler, while studious ones draw flak. Look at the sidebar "Highlights of the `O(1)` Scheduler" to find out some of its important features.

Highlights of the `O(1)` Scheduler

The following are some of the important features of the `O(1)` scheduler:

- The algorithm uses two *run queues* made up of 140 priority lists: an *active* queue that holds tasks that have time slices left and an *expired* queue that contains processes whose time slices have expired. When a task finishes its time slice, it's inserted into the expired queue in sorted order of priority. The active and expired queues are swapped when the former becomes empty. To decide which process to run next, the scheduler does not navigate through the entire queue. Instead, it picks that task from the active queue having the highest priority. The overhead of picking the task thus depends not on the number of active tasks, but on the number of priorities. This makes it a constant-time or an `O(1)` algorithm.

- The scheduler supports two priority ranges: standard *nice*values supported on UNIX systems and internal priorities. The former range from –20 to +19, while the latter extend from 0 to 139. In both cases, lower values correspond to higher priorities. The top 100 (0 to 99) internal priorities are reserved for real time (RT) tasks, and the bottom 40 (100 to 139) are assigned to normal tasks. The 40 *nice* values map to the bottom 40 internal priorities. Internal priorities of normal tasks can be dynamically varied by the scheduler, whereas *nice* values are statistically set by the user. Each internal priority gets an associated run list.

- The scheduler uses a heuristic to figure out whether the nature of a process is I/O-intensive or CPU-intensive. In simple terms, if a task sleeps often, it's likely to be I/O-intensive, but if it uses its time slice fast, it's CPU-intensive. Whenever the scheduler finds that a task has demonstrated I/O-bound characteristics, it rewards it by dynamically increasing its internal priority. CPU-bound characteristics, on the other hand, are punished with negative marks.

- The allotted time slice is directly proportional to the priority. A higher priority task gets a bigger time slice.

- A task will not be preempted by the scheduler as long as it has time slice credit. If it yields the processor before using up its time slice quota, it can roll over the reminder of its slice when it's run next. Because I/O-bound processes are the ones that often yield the CPU, this improves interactive performance.

- The scheduler supports RT scheduling policies. RT tasks preempt normal (`SCHED_OTHER`) tasks. Users of RT policies can override the scheduler's dynamic priority assignments. Unlike `SCHED_OTHER` tasks, their priorities are not recalculated by the kernel on-the-fly. RT scheduling comes in two flavors: `SCHED_FIFO` and `SCHED_RR`. They are used for producing "soft" real-time behavior, rather than stringent "hard" RT guarantees. `SCHED_FIFO` has no concept of time slices; `SCHED_FIFO` tasks run until they sleep-wait for I/O or yield the processor. `SCHED_RR` is a round-robin variant of `SCHED_FIFO` that also assigns time slices to RT tasks. `SCHED_RR` tasks with expired slices are appended to the end of the corresponding priority list.

- The scheduler improves SMP performance by using per-CPU run queues and per-CPU synchronization.

The CFS Scheduler

The Linux scheduler has undergone another rewrite with the 2.6.23 kernel. The *Completely Fair Scheduler* (CFS) for the `SCHED_OTHER` class removes much of the complexities associated with the `O(1)` scheduler by abandoning priority arrays, time slices, interactivity heuristics, and the dependency on `HZ`. CFS's goal is to implement fairness for all scheduling entities by providing each task the total CPU power divided by the number of running tasks. Dissecting CFS is beyond the scope of this chapter. Have a look at *Documentation/sched-design-CFS.txt* for a brief tutorial.

Response Times

As a user mode driver developer, you have several options to improve your application's response time:

- Use RT scheduling policies that give you a finer grain of control than usual. Look at the man pages of `sched_setscheduler()` and its relatives for insights into achieving soft RT response times.

- If you are using non-RT scheduling, tune the *nice* values of different processes to achieve the required performance balance.

- If you are using a 2.6.23 or later kernel enabled with the CFS scheduler, you may fine-tune */proc/sys/kernel/sched_granularity_ns.* If you are using a pre-2.6.23 kernel, modify `#defines` in *kernel/sched.c* and *include/linux/sched.h* to suit your application. Change these values cautiously to satisfy the needs of your application suite. Usage scenarios of the scheduler are complex. Settings that delight certain load conditions can depress others, so you may have to experiment by trial and error.

- Response times are not solely the domain of the scheduler; they also depend on the solution architecture. For example, if you mark a busy interrupt handler as *fast*, it disables other local interrupts frequently and that can potentially slow down data acquisition and transmission on other IRQs.

Let's implement an example and see how a user mode driver can achieve fast response times by guarding against indeterminism introduced by scheduling and paging. As you learned in Chapter 2, "A Peek Inside the Kernel," the RTC is a timer source that can generate periodic interrupts with high precision. Listing 19.1 implements an example that uses interrupt reports from */dev/rtc* to perform periodic work with microsecond precision. The Pentium *Time Stamp Counter* (TSC) is used to measure response times.

The program in Listing 19.1 first changes its scheduling policy to `SCHED_FIFO` using `sched_setscheduler()`. Next, it invokes `mlockall()` to lock all mapped pages in memory to ensure that swapping won't come in the way of deterministic timing. Only the super-user is allowed to call `sched_setscheduler()`and `mlockall()` and request RTC interrupts at frequencies greater than 64Hz.

Listing 19.1. Periodic Work with Microsecond Precision

Code View:

```
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <fcntl.h>
#include <pthread.h>
#include <linux/mman.h>

/* Read the lower half of the Pentium Time Stamp Counter
   using the rdtsc instruction */
#define rdtscl(val) __asm__ __volatile__ ("rdtsc" : "=A" (val))


main()
{
  unsigned long ts0, ts1, now, worst; /* Store TSC ticks */
  struct sched_param sched_p;         /* Information related to
                                         scheduling priority */
  int fd, i=0;
  unsigned long data;
 /* Change the scheduling policy to SCHED_FIFO */
 sched_getparam(getpid(), &sched_p);
 sched_p.sched_priority = 50; /* RT Priority */
 sched_setscheduler(getpid(), SCHED_FIFO, &sched_p);

 /* Avoid paging and related indeterminism */
 mlockall(MCL_CURRENT);

 /* Open the RTC */
 fd = open("/dev/rtc", O_RDONLY);

 /* Set the periodic interrupt frequency to 8192Hz
    This should give an interrupt rate of 122uS */
 ioctl(fd, RTC_IRQP_SET, 8192);

 /* Enable periodic interrupts */
 ioctl(fd, RTC_PIE_ON, 0);
 rdtscl(ts0);
 worst = 0;

 while (i++ < 10000) {

   /* Block until the next periodic interrupt */
   read(fd, &data, sizeof(unsigned long));
```

```
   /* Use the TSC to precisely measure the time consumed.
      Reading the lower half of the TSC is sufficient */
   rdtscl(ts1);
   now = (ts1-ts0);

   /* Update the worst case latency */
   if (now > worst) worst = now;
   ts0 = ts1;

   /* Do work that is to be done periodically */
   do_work(); /* NOP for the purpose of this measurement */
 }

 printf("Worst latency was %8ld\n", worst);

 /* Disable periodic interrupts */
 ioctl(fd, RTC_PIE_OFF, 0);
}
```

The code in Listing 19.1 loops for 10,000 iterations and prints out the worst-case latency that occurred during execution. The output was 240899 on a Pentium 1.8GHz box, which roughly corresponds to 133 microseconds. According to the data sheet of the RTC chipset, a timer frequency of 8192Hz should result in a periodic interrupt rate of 122 microseconds. That's close. Rerun the code under varying loads using `SCHED_OTHER` instead of `SCHED_FIFO` and observe the resultant drift.

You may also run kernel threads in the RT mode. For that, do the following when you start the thread:

```
static int
my_kernel_thread(void *i)
{
  daemonize();
  current->policy = SCHED_FIFO;
  current->rt_priority = 1;
  /* ... */
}
```

Chapter 19. Drivers in User Space

In This Chapter

- Process Scheduling and Response Times 553
- Accessing I/O Regions 558
- Accessing Memory Regions 562
- User Mode SCSI 565
- User Mode USB 567
- User Mode I^2C 571
- UIO 573
- Looking at the Sources 574

Most device drivers prefer to lead a privileged life inside the kernel, but some are at home in the indeterministic world outside. Several kernel subsystems, such as SCSI, USB, and I^2C, offer some level of support for user mode drivers, so you might be able to control those devices without writing a single line of kernel code.

In spite of the inclement weather in user land, user mode drivers enjoy certain advantages. They are easy to develop and debug. You won't have to reboot the system every time you dereference a dangling pointer. Some user mode drivers will even work across operating systems if the device subsystem enjoys the services of a standard user-space programming library. Here are some rules of thumb to help decide whether your driver should reside in user space:

- Apply the possibility test. What can be done in user space should probably stay in user space.
- If you have to talk to a large number of slow devices and if performance requirements are modest, explore the possibility of implementing the drivers in user space. If you have time-critical performance requirements, stay inside the kernel.
- If your code needs the services of kernel APIs, access to kernel variables, or is intertwined with interrupt handling, it has a strong case for being in kernel space.
- If much of what your code does can be construed as policy, user land might be its logical residence.
- If the rest of the kernel needs to invoke your code's services, it's a candidate for staying inside the kernel.
- You can't easily do floating-point arithmetic inside the kernel. *Floating-point unit* (FPU) instructions can, however, be used from user space.

That said, you can't accomplish too much from user space. Many important device classes, such as storage media and network adapters, cannot be driven from user land. But even when a kernel driver is the appropriate solution, it's a good idea to model and test as much code as you can in user space before moving it to kernel space. The testing cycle is faster, and it's easier to traverse all possible code paths and ensure that they are clean.

In this chapter, the term *user space driver* (or *user mode driver*) is used in a generic sense that does not strictly conform to the semantics of a driver implied thus far in the book. An application is considered to be a user mode driver if it's a candidate for being implemented inside the kernel, too.

The 2.6 kernel overhauled a subsystem that is of special interest to user space drivers. The new process scheduler offers huge response-time benefits to user mode code, so let's start with that.

Process Scheduling and Response Times

Many user mode drivers need to perform some work in a time-bound manner. In user space, indeterminism due

to scheduling and paging often come in the way of fast response times, however. To see how you can minimize the impact of the former hurdle, let's dip into recent Linux schedulers and understand their underlying philosophy.

The Original Scheduler

In the 2.4 and earlier days, the scheduler used to recalculate scheduling parameters of each task before taking its pick. The time consumed by the algorithm thus increased linearly with the number of contending tasks in the system. In other words, it used `O(n)` time, where `n` is the number of active tasks. On a system running at high loads, this translated to significant overhead. The 2.4 algorithm also didn't work very well on SMP systems.

The O(1) Scheduler

Time consumed by an `O(n)` algorithm depends linearly on the size of its input, and an $O(n^2)$ solution depends quadratically on the length of its input, but an `O(1)` technique is independent of the input and thus scales well. The 2.6 scheduler replaced the `O(n)` algorithm with an `O(1)` method. In addition to being super-scalable, the scheduler has built-in heuristics to improve user responsiveness by providing preferential treatment to tasks involved in I/O activity. Processes are of two kinds: I/O bound and CPU bound. I/O-bound tasks are often sleep-waiting for device I/O, while CPU-bound ones are workaholics addicted to the processor. Paradoxically, to achieve fast response times, lazy tasks get incentives from the `O(1)` scheduler, while studious ones draw flak. Look at the sidebar "Highlights of the `O(1)` Scheduler" to find out some of its important features.

Highlights of the `O(1)` Scheduler

The following are some of the important features of the `O(1)` scheduler:

- The algorithm uses two *run queues* made up of 140 priority lists: an *active* queue that holds tasks that have time slices left and an *expired* queue that contains processes whose time slices have expired. When a task finishes its time slice, it's inserted into the expired queue in sorted order of priority. The active and expired queues are swapped when the former becomes empty. To decide which process to run next, the scheduler does not navigate through the entire queue. Instead, it picks that task from the active queue having the highest priority. The overhead of picking the task thus depends not on the number of active tasks, but on the number of priorities. This makes it a constant-time or an `O(1)` algorithm.

- The scheduler supports two priority ranges: standard *nice*values supported on UNIX systems and internal priorities. The former range from –20 to +19, while the latter extend from 0 to 139. In both cases, lower values correspond to higher priorities. The top 100 (0 to 99) internal priorities are reserved for real time (RT) tasks, and the bottom 40 (100 to 139) are assigned to normal tasks. The 40 *nice* values map to the bottom 40 internal priorities. Internal priorities of normal tasks can be dynamically varied by the scheduler, whereas *nice* values are statistically set by the user. Each internal priority gets an associated run list.

- The scheduler uses a heuristic to figure out whether the nature of a process is I/O-intensive or CPU-intensive. In simple terms, if a task sleeps often, it's likely to be I/O-intensive, but if it uses its time slice fast, it's CPU-intensive. Whenever the scheduler finds that a task has demonstrated I/O-bound characteristics, it rewards it by dynamically increasing its internal priority. CPU-bound characteristics, on the other hand, are punished with negative marks.

- The allotted time slice is directly proportional to the priority. A higher priority task gets a bigger time slice.

- A task will not be preempted by the scheduler as long as it has time slice credit. If it yields the processor before using up its time slice quota, it can roll over the reminder of its slice when it's run next. Because I/O-bound processes are the ones that often yield the CPU, this improves interactive performance.

- The scheduler supports RT scheduling policies. RT tasks preempt normal (`SCHED_OTHER`) tasks. Users of RT policies can override the scheduler's dynamic priority assignments. Unlike `SCHED_OTHER` tasks, their priorities are not recalculated by the kernel on-the-fly. RT scheduling comes in two flavors: `SCHED_FIFO` and `SCHED_RR`. They are used for producing "soft" real-time behavior, rather than stringent "hard" RT guarantees. `SCHED_FIFO` has no concept of time slices; `SCHED_FIFO` tasks run until they sleep-wait for I/O or yield the processor. `SCHED_RR` is a round-robin variant of `SCHED_FIFO` that also assigns time slices to RT tasks. `SCHED_RR` tasks with expired slices are appended to the end of the corresponding priority list.

- The scheduler improves SMP performance by using per-CPU run queues and per-CPU synchronization.

The CFS Scheduler

The Linux scheduler has undergone another rewrite with the 2.6.23 kernel. The *Completely Fair Scheduler* (CFS) for the `SCHED_OTHER` class removes much of the complexities associated with the `O(1)` scheduler by abandoning priority arrays, time slices, interactivity heuristics, and the dependency on `HZ`. CFS's goal is to implement fairness for all scheduling entities by providing each task the total CPU power divided by the number of running tasks. Dissecting CFS is beyond the scope of this chapter. Have a look at *Documentation/sched-design-CFS.txt* for a brief tutorial.

Response Times

As a user mode driver developer, you have several options to improve your application's response time:

- Use RT scheduling policies that give you a finer grain of control than usual. Look at the man pages of `sched_setscheduler()` and its relatives for insights into achieving soft RT response times.

- If you are using non-RT scheduling, tune the *nice* values of different processes to achieve the required performance balance.

- If you are using a 2.6.23 or later kernel enabled with the CFS scheduler, you may fine-tune */proc/sys/kernel/sched_granularity_ns.* If you are using a pre-2.6.23 kernel, modify `#defines` in *kernel/sched.c* and *include/linux/sched.h* to suit your application. Change these values cautiously to satisfy the needs of your application suite. Usage scenarios of the scheduler are complex. Settings that delight certain load conditions can depress others, so you may have to experiment by trial and error.

- Response times are not solely the domain of the scheduler; they also depend on the solution architecture. For example, if you mark a busy interrupt handler as *fast*, it disables other local interrupts frequently and that can potentially slow down data acquisition and transmission on other IRQs.

Let's implement an example and see how a user mode driver can achieve fast response times by guarding against indeterminism introduced by scheduling and paging. As you learned in Chapter 2, "A Peek Inside the Kernel," the RTC is a timer source that can generate periodic interrupts with high precision. Listing 19.1 implements an example that uses interrupt reports from */dev/rtc* to perform periodic work with microsecond precision. The Pentium *Time Stamp Counter* (TSC) is used to measure response times.

The program in Listing 19.1 first changes its scheduling policy to `SCHED_FIFO` using `sched_setscheduler()`. Next, it invokes `mlockall()` to lock all mapped pages in memory to ensure that swapping won't come in the way of deterministic timing. Only the super-user is allowed to call `sched_setscheduler()`and `mlockall()` and request RTC interrupts at frequencies greater than 64Hz.

Listing 19.1. Periodic Work with Microsecond Precision

Code View:

```
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <fcntl.h>
#include <pthread.h>
#include <linux/mman.h>

/* Read the lower half of the Pentium Time Stamp Counter
   using the rdtsc instruction */
#define rdtscl(val) __asm__ __volatile__ ("rdtsc" : "=A" (val))


main()
{
  unsigned long ts0, ts1, now, worst; /* Store TSC ticks */
  struct sched_param sched_p;         /* Information related to
                                         scheduling priority */
  int fd, i=0;
  unsigned long data;
 /* Change the scheduling policy to SCHED_FIFO */
 sched_getparam(getpid(), &sched_p);
 sched_p.sched_priority = 50; /* RT Priority */
 sched_setscheduler(getpid(), SCHED_FIFO, &sched_p);

 /* Avoid paging and related indeterminism */
 mlockall(MCL_CURRENT);

 /* Open the RTC */
 fd = open("/dev/rtc", O_RDONLY);

 /* Set the periodic interrupt frequency to 8192Hz
    This should give an interrupt rate of 122uS */
 ioctl(fd, RTC_IRQP_SET, 8192);

 /* Enable periodic interrupts */
 ioctl(fd, RTC_PIE_ON, 0);
 rdtscl(ts0);
 worst = 0;

 while (i++ < 10000) {

   /* Block until the next periodic interrupt */
   read(fd, &data, sizeof(unsigned long));
```

```
   /* Use the TSC to precisely measure the time consumed.
      Reading the lower half of the TSC is sufficient */
   rdtscl(ts1);
   now = (ts1-ts0);

   /* Update the worst case latency */
   if (now > worst) worst = now;
   ts0 = ts1;

   /* Do work that is to be done periodically */
   do_work(); /* NOP for the purpose of this measurement */
 }

 printf("Worst latency was %8ld\n", worst);

 /* Disable periodic interrupts */
 ioctl(fd, RTC_PIE_OFF, 0);
}
```

The code in Listing 19.1 loops for 10,000 iterations and prints out the worst-case latency that occurred during execution. The output was 240899 on a Pentium 1.8GHz box, which roughly corresponds to 133 microseconds. According to the data sheet of the RTC chipset, a timer frequency of 8192Hz should result in a periodic interrupt rate of 122 microseconds. That's close. Rerun the code under varying loads using `SCHED_OTHER` instead of `SCHED_FIFO` and observe the resultant drift.

You may also run kernel threads in the RT mode. For that, do the following when you start the thread:

```
static int
my_kernel_thread(void *i)
{
  daemonize();
  current->policy = SCHED_FIFO;
  current->rt_priority = 1;
  /* ... */
}
```

Accessing I/O Regions

PC-compatible systems have 64K I/O ports, all of which may be driven from user space. User access to I/O ports on Linux is controlled by two functions: `ioperm()` and `iopl()`. `ioperm()` controls access permissions to the first `0x3ff` ports. `iopl()` changes the I/O privilege level of the calling process, thus allowing among other things, unrestricted access to all ports. Only the super-user can invoke both these functions.

To write data to an I/O port, use `outb()`, `outw()`, `outl()`, or their cousins. To read data from a port, use `inb()`, `inw()`, `inl()`, or their relatives. Let's implement a simple program that reads the seconds ticking inside the RTC chip. I/O regions in the PC CMOS, of which the RTC is a part, are accessed via an index port (`0x70`) and a data port (`0x71`), as shown in Table 5.1 of Chapter 5, "Character Drivers." To read a byte of data from offset `off` within an I/O address range, write `off` to the index port and read the associated data from the data port. Listing 19.2 reads the seconds field of the RTC; but to use it to obtain data from other I/O regions, change the arguments passed to `dump_port()` suitably.

Listing 19.2. Utility to Dump Bytes from an I/O Region

Code View:

```
#include <linux/ioport.h>

void
dump_port(unsigned char addr_port, unsigned char data_port,
          unsigned short offset, unsigned short length)
{
  unsigned char i, *data;

  if (!(data = (unsigned char *)malloc(length))) {
    perror("Bad Malloc\n");
    exit(1);
  }

  /* Write the offset to the index port
     and read data from the data port */
  for(i=offset; i<offset+length; i++) {
    outb(i, addr_port  );
    data[i-offset] = inb(data_port);
  }

  /* Dump */
  for(i=0; i<length; i++)
    printf("%02X ", data[i]);

  free(data);
}

int
main(int argc, char *argv[])
{

  /* Get access permissions */
  if( iopl(3) < 0)  {
    perror("iopl access error\n");
    exit(1);
  }
```

```
  dump_port(0x70, 0x71, 0x0, 1);
}
```

You may also accomplish the same task by operating on */dev/port.* This will incur a performance penalty because code flow has to pass through a kernel driver, but you have the flexibility to control access permissions on the device node without using `iopl()` or `ioperm()`. Here's the */dev/port* equivalent of Listing 19.2:

```
#include <unistd.h>
#include <fcntl.h>

int
main(int argc, char *argv[])
{
  char seconds=0;
  char data = 0;
  int fd = open("/dev/port", O_RDWR);

  lseek(fd, 0x70, SEEK_SET);
  write(fd, &data, 1);

  lseek(fd, 0x71, SEEK_SET);
  read(fd, &seconds, 1);
  printf("%02X ", seconds);
}
```

In Chapter 5, you learned to talk to your computer's parallel port via a kernel driver. Let's now implement a sample program that interacts with a parallel port device from user space. The kernel's parallel port subsystem provides a character driver called *ppdev* that exports parallel port access to user land. Ppdev creates device nodes, */dev/parportX*, where *X* is the parallel port number. Applications can open */dev/parportX*, exchange data via `read()`/`write()` system calls, and issue a variety of `ioctl()` commands. Using kernel interfaces, such as ppdev, is preferable to directly operating over I/O ports using `ioperm()`, `iopl()`, or */dev/port.* The former technique is safer, works across architectures, and functions over different device form factors such as USB-to-parallel converters.

Consider the simple LED board that you used in Chapter 5. It had 8 LEDs interfaced to pins 2 to 9 on a standard 25-pin parallel connector. Listing 19.3 implements a simple user application that glows alternate diodes on this parallel port LED board using the ppdev interface. It's the user-space equivalent of the kernel driver developed in Listing 5.6 of Chapter 5.

Listing 19.3. Controlling a Parallel Port LED Board from User Space

Code View:

```
#include <stdio.h>
#include <linux/ioctl.h>
#include <linux/parport.h>
#include <linux/ppdev.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
  int led_fd;
  char data = 0xAA; /* Bit pattern to glow alternate LEDs */

  /* Open /dev/parport0. This assumes that the LED connector board
     is connected to the first parallel port on your computer */
  if ((led_fd = open("/dev/parport0", O_RDWR)) < 0) {
    perror("Bad Open\n");
    exit(1);
  }

  /* Claim the port */
  if (ioctl(led_fd, PPCLAIM)) {
    perror("Bad Claim\n");
    exit(2);
  }

  /* Set pins to forward direction and write a
     byte to glow alternate LEDs */
  if (ioctl(led_fd, PPWDATA, &data)) {
    perror("Bad Write\n");
    exit(3);
  }

  /* Release the port */
  if (ioctl(led_fd, PPRELEASE)) {
    perror("Bad Release\n");
    exit(4);
  }

  /* Close /dev/parport0 */
  close(led_fd);
}
```

Accessing Memory Regions

Memory mapping (or *mmaping*) a file associates it with an area of user virtual memory. Because Linux treats devices as files, you can also map device memory to RAM and directly operate on it from user space. Here are some `mmap()` users on Linux:

1. Graphical user interfaces, such as X Windows (www.xfree86.org) and SVGAlib (www.svgalib.org), mmap video memory and directly access graphics hardware.

2. *Madplay* is an integer-only MP3 player that runs on several architectures. Memory mapping improves throughput, so madplay mmaps MP3 files for faster access. This helps maintain the correct bit rates necessary for high-quality music playback.

3. MPEG (*Moving Picture Experts Group*) decoders play movies by directly operating on mmapped frame buffer memory.

The prototype of the `mmap()` system call looks like this:

```
void *mmap(void *start, size_t length, int prot, int flag,
           int fd, off_t offset);
```

This requests the kernel to associate the device file specified by the file descriptor `fd` to a chunk of user memory beginning at `start`. (`start` is only a preference and is usually set to `0`; the actual associated memory is returned by `mmap()`.) The kernel maps `length` bytes of memory starting from `offset` in the specified file. `prot` specifies the desired access protection, and `flag` describes the type of the mapping. The `MAP_SHARED` flag mirrors your modifications to other users of the same memory region, whereas `MAP_PRIVATE` keeps your changes to yourself.

All mmapped pages need not be present in physical memory. Areas not being accessed can be in swap space from where they are paged in on demand. Underlying device drivers may control the semantics of the `mmap()` system call by implementing an `mmap()` method.

Listing 19.4 is an image display program that illustrates usage of `mmap()` as follows:

- Mmaps a frame buffer. (We discussed frame buffer drivers in Chapter 12, "Video Drivers.")

- Mmaps an image file.

- Transfers the latter to the former after performing necessary transformations depending on the properties of the image file (not shown in the listing).

Listing 19.4. Displaying an Image Using Mmap

Code View:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>  /* For definition of mmap() */
#include <linux/fb.h>  /* For frame buffer structures and ioctls */


int
main(int argc, char *argv[])
{
  int imagefd, fbfd;                  /* File descriptors */
  char *imagebuf, *fbbuf;             /* mmap buffers */
  struct fb_var_screeninfo vinfo;     /* Variable Screen info */
  struct stat statbuf;                /* Image info */
  int fbsize;                         /* Frame buffer size */

  /* Open image file */
  if ((imagefd = open(argv[1], O_RDONLY)) < 0) {
    perror("Bad image open\n");
    exit(1);
  }

  /* Get the size of the image file */
  if (fstat(imagefd, &statbuf) < 0) {
    perror("Bad fstat\n");
    exit(1);
  }


  /* mmap the image file */
  if ((imagebuf = mmap(0, statbuf.st_size, PROT_READ, MAP_SHARED,
                       imagefd, 0)) == (char *) -1){
    perror("Bad image mmap\n");
    exit(1);
  }
  /* Open video memory */
  if ((fbfd = open("/dev/fb0", O_RDWR)) < 0) {
    perror("Bad frame buffer open\n");
    exit(1);
  }

  /* Get screen attributes such as resolution and depth */
  if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
    perror("Bad vscreeninfo ioctl\n");
    exit(1);
  }

  /* Size of video memory =
     (X-resolution * Y-resolution * Bytes per pixel) */
  fbsize = (vinfo.xres * vinfo.yres * vinfo.bits_per_pixel)/8;

  /* mmap the video memory */
  if ((fbbuf = mmap(0, fbsize, PROT_WRITE, MAP_SHARED, fbfd, 0))
      == (char *) -1){
    perror("Bad frame buffer mmap\n");
    exit(1);
  }

  /* Transfer imagebuf to fbbuf after applying transformations
     dependent on the format, resolution, depth, data offset,
```

```
     and other properties of the image file. Not implemented in
     this listing */
  copy_image_to_fb();

  msync(fbbuf, fbsize, MS_SYNC); /* Flush changes to device */

  /* ... */

  /* Unmap frame buffer memory */
  munmap(fbbuf, fbsize);
  close(fbfd);

  /* Unmap image file */
  munmap(imagebuf, statbuf.st_size);
  close(imagefd);

}
```

User Mode SCSI

The *SCSI Generic* (*sg*) interface lets you directly dispatch SCSI commands from user space. The *sg* driver essentially exports a char interface, so applications can use the `open()`, `close()`, `read()`, `write()`, `ioctl()`, `poll()`, `fcntl()`, and `mmap()` system calls to talk to the underlying device. Drivers for SCSI devices such as CD burners and scanners are implemented in user space over sg. Look at the sources of *cdrtools* (previously called *cdrecord*) available from http://freshmeat.net/projects/cdrecord/ for a real-life sg user.

Let's learn how to use the sg interface with the help of an example. Listing 19.5 implements a user program that sends a `READ_CAPACITY` SCSI command to a storage device, such as a SCSI hard disk or a USB mass storage drive to glean its data capacity. The `READ_CAPACITY` command consists of 10 bytes, starting with the command code `0x25`. For the purpose of this example, let's set the rest of the bytes to zero. When a SCSI device receives a `READ_CAPACITY` command, it responds with an 8-byte reply; the top 4 bytes contain the address of the last logical block, and the bottom 4 bytes contain the block length.

sg device nodes are named */dev/sgX*, where *X* is the device number, so Listing 19.5 opens */dev/sg0*, which is assumed to be the sg char node corresponding to your SCSI storage device. It then sets about populating the `sg_io_hdr_t` structure, which is the main data structure that sg users have to manage. The `read()`, `write()`, and `ioctl()`calls expect a pointer to this structure (defined in */usr/include/scsi/sg.h*) as an argument. The `cmdp` field of `sg_io_hdr_t` is set to the address of the command block that holds the 10-byte `READ_CAPACITY` command. The `dxferp` field supplies the address of a buffer that will carry the response data arriving from the device. The `sbp` field contains the address of a *sense* buffer that will return the status of the requested operation. The `interface_id` has to be set to `S`, and `timeout` holds the wait time in milliseconds before sg gives up on the command.

`SG_IO` is an oft-used ioctl command supported by sg. Internally, it writes a command to the device, waits for a response, and reads the received reply into a user-supplied buffer. In Listing 19.5, `SG_IO` issues a `READ_CAPACITY` command and reads the 8-byte response into `rcap_buff[]`. The program calculates and prints the disk capacity by interpreting the data in `rcap_buff[]`.

Listing 19.5. Obtaining Disk Capacity via SCSI Generic

Code View:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <scsi/sg.h>

#define RCAP_COMMAND     0x25
#define RCAP_COMMAND_LEN 10
#define RCAP_REPLY_LEN   8

int
main(int argc, char *argv[])
{
  int fd, i;
  /* READ_CAPACITY command block */
  unsigned char RCAP_CmdBlk[RCAP_COMMAND_LEN]=
                              {RCAP_COMMAND, 0,0,0,0,0,0,0,0,0};
  sg_io_hdr_t sg_io;
  unsigned char rcap_buff[RCAP_REPLY_LEN];
  unsigned int lastblock, blocksize;
  unsigned long long disk_cap;
```

```
  unsigned char sense_buf[32];

  /* Open the sg device */
  if ((fd = open("/dev/sg0", O_RDONLY)) < 0) {
    printf("Bad Open\n");
    exit(1);
  }

  /* Initialize */
  memset(&sg_io, 0, sizeof(sg_io_hdr_t));

  /* Command block address and length */
  sg_io.cmdp = RCAP_CmdBlk;
  sg_io.cmd_len = RCAP_COMMAND_LEN;

  /* Response buffer address and length */
  sg_io.dxferp = rcap_buff;
  sg_io.dxfer_len = RCAP_REPLY_LEN;

  /* Sense buffer address and length */
  sg_io.sbp = sense_buf;
  sg_io.mx_sb_len = sizeof(sense_buf);
  /* Control information */
  sg_io.interface_id = 'S';
  sg_io.dxfer_direction = SG_DXFER_FROM_DEV;
  sg_io.timeout = 10000; /* 10 seconds */

  /* Issue the SG_IO ioctl */
  if (ioctl(fd, SG_IO, &sg_io) < 0) {
    printf("Bad SG_IO\n");
    exit(1);
  }

  /* Obtain results */
  if ((sg_io.info & SG_INFO_OK_MASK) == SG_INFO_OK) {
    /* Address of last disk block */
    lastblock =  ((rcap_buff[0]<<24)|(rcap_buff[1]<<16)|
              (rcap_buff[2]<<8)|(rcap_buff[3]));

    /* Block size */
    blocksize =  ((rcap_buff[4]<<24)|(rcap_buff[5]<<16)|
              (rcap_buff[6]<<8)|(rcap_buff[7]));

    /* Calculate disk capacity */
    disk_cap  = (lastblock+1);
    disk_cap *= blocksize;
    printf("Disk Capacity = %llu Bytes\n", disk_cap);

  }
  close(fd);
}
```

For the full list of SG_IO commands, take a look at *include/scsi/scsi.h* and *drivers/scsi/sg.c*. Read the Linux SCSI Generic HOWTO for an in-depth explanation of the sg interface. Download the *sg3_utils* package from http://sg.torque.net/sg/sg3_utils.html and browse the sources to find several useful programs that operate over sg.

User Mode USB

The *usbfs* virtual filesystem allows raw access to USB devices from user space. Usbfs is usually mounted over */proc/bus/usb/.* The usbfs tree contains directories corresponding to each USB controller (or bus) on your system. Each of these directories, in turn, contains nodes corresponding to USB devices present on that bus.

To better understand usbfs, let's look at a system with an Intel ICH4 South Bridge chipset. As you learned in Chapter 11, "Universal Serial Bus," USB controllers are part of the South Bridge chipset on PC systems. The ICH4 supports one USB EHCI (high-speed USB 2.0) controller and three USB UHCI controllers and can connect to six physical USB ports. The EHCI controller can converse with all six ports, and the three UHCI controllers can talk to two ports each. Let's call the EHCI controller *bus1* and the three UHCI controllers *bus2, bus3*, and *bus4*, respectively. Now assume that the system has only two physical USB ports and that they are connected to the UHCI controller corresponding to bus3. (The → symbol attaches comments to command output.)

Code View:
```
bash> ls –lR /proc/bus/usb
/proc/bus/usb:
total 0
dr-xr-xr-x  2 root root 0 Dec  2 12:44 001    → EHCI. Can talk to
                                                 any physical port
dr-xr-xr-x  2 root root 0 Dec  2 12:44 002    → No corresponding
                                                 physical ports
dr-xr-xr-x  2 root root 0 Dec  2 12:44 003    → UHCI bus for the 2
                                                 physical USB ports
                                                on this system
dr-xr-xr-x  2 root root 0 Dec  2 12:44 004    → No corresponding
                                                 physical ports
-r--r--r--  1 root root 0 Dec  2 20:02 devices

/proc/bus/usb/001:
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001   → Root Hub (bus1)

/proc/bus/usb/002:
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001   → Root Hub (bus2)

/proc/bus/usb/003:
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001   → Root Hub (bus3)

/proc/bus/usb/004:
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001   → Root Hub (bus4)
```

Let's connect a full-speed Nikon digital camera and a high-speed Seagate USB 2.0 hard disk to the two USB ports on the system. First, take a peek at */proc/bus/usb/devices* and find the relevant entries:

Code View:
```
bash> ls –lR /proc/bus/usb/devices
```

```
...
T:  Bus=03 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  5 Spd=12  MxCh= 0
D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=04b0 ProdID=0205 Rev= 1.00
S:  Manufacturer=NIKON
S:  Product=NIKON DSC E5200
S:  SerialNumber=2507597
C:* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr=  2mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50
    Driver=usb-storage
E:  Ad=01(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=82(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
...
T:  Bus=01 Lev=01 Prnt=01 Port=02 Cnt=01 Dev#= 12 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0bc2 ProdID=0501 Rev= 0.01
S:  Manufacturer=Seagate
S:  Product=USB Mass Storage
S:  SerialNumber=000000062459
C:* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr= 0mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50
    Driver=usb-storage
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=88(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
```

Look at the `T:` lines in the preceding output, which display the topology. As expected, the hard disk has arrived on the EHCI bus, *bus1*, and the camera has appeared on the UHCI bus, *bus3*. This is how the usbfs tree looks now:

Code View:

```
bash> ls -lR /proc/bus/usb
/proc/bus/usb:
total 0
dr-xr-xr-x  2 root root 0 Dec  2 12:44 001
dr-xr-xr-x  2 root root 0 Dec  2 12:44 002
dr-xr-xr-x  2 root root 0 Dec  2 12:44 003
dr-xr-xr-x  2 root root 0 Dec  2 12:44 004
-r--r--r--  1 root root 0 Dec  2 19:51 devices
/proc/bus/usb/001:                          ⟶   EHCI: bus1
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001
-rw-r--r--  1 root root 50 Dec  2 19:51 007 ⟶   High-speed disk

/proc/bus/usb/002:                          ⟶   UHCI: bus2
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001

/proc/bus/usb/003:                          ⟶   UHCI: bus3
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001
-rw-r--r--  1 root root 50 Dec  2 19:16 003 ⟶   Full-speed camera
```

```
/proc/bus/usb/004:                          →   UHCI: bus4
total 0
-rw-r--r--  1 root root 43 Dec  2 12:44 001
```

The usbfs files corresponding to plugged-in devices contain the associated USB device and configuration descriptors. In the preceding example, read */proc/bus/usb/003/003* to get descriptor information for the camera and */proc/bus/usb/001/007* for the descriptor associated with the hard disk. Managing usbfs files is not straightforward however, because the device filenames get reused after a device is plugged out. The solution is to use the *libusb* library, which uses usbfs under the hood. Using libusb instead of directly operating on usbfs has another benefit: Your driver is likely to work unchanged on other operating systems that support this library. If you don't find libusb bundled along with your distribution, download its sources from http://libusb.sourceforge.net/. The full list of USB access functions offered by this library is available under the *doc/* directory of the libusb sources.

Listing 19.6 implements a skeletal user space driver for the digital camera using an oft-used libusb programming template. The camera's vendor ID (`0x04b0`) and device ID (`0x0205`) are obtained from the */proc/bus/usb/devices* output shown previously.

Listing 19.6. A Skeletal User Space USB Driver Using libusb

Code View:

```
#include <usb.h>                    /* From the libusb package */

#define DIGICAM_VENDOR_ID   0x04b0 /* From /proc/bus/usb/devices */
#define DIGICAM_PRODUCT_ID  0x0205 /* From /proc/bus/usb/devices */
int
main(int argc, char *argv[])
{
  struct usb_dev_handle *mydevice_handle;
  struct usb_bus *usb_bus;
  struct usb_device *mydevice;

  /* Initialize libusb */
  usb_init();
  usb_find_buses();
  usb_find_devices();

  /* Walk the bus */
  for (usb_bus = usb_buses; usb_bus; usb_bus = usb_bus->next) {
    for (mydevice = usb_bus->devices; mydevice;
         mydevice = mydevice->next) {
      if ((mydevice->descriptor.idVendor == DIGICAM_VENDOR_ID) &&
       (mydevice->descriptor.idProduct == DIGICAM_PRODUCT_ID)) {

        /* Open the device */
        mydevice_handle = usb_open(mydevice);

        /* Send commands to the camera. This is the heart of the
           driver. Getting information about the USB control
           messages to which your device responds is often a
           challenge since many vendors do not readily divulge
           hardware details */
        usb_control_msg(mydevice_handle, ...);
        /* ... */
```

```
        /* Close the device */
        usb_close(mydevice_handle);
      }
    }
  }
}
```

User Mode I²C

In Chapter 8, "The Inter-Integrated Circuit Protocol," you learned to develop kernel mode drivers for I²C devices; but sometimes, when you need to enable support for a large number of slow I²C devices, it makes sense to drive them from user space. The *i2c-dev* module enables the development of user mode I²C/SMBus device drivers. User-space code can access I²C host adapters via device nodes. To operate on the n^{th} adapter, open */dev/i2c-n.* After you have a file descriptor tied to a host adapter device node, you can command it through ioctls to connect to specific slave devices attached to it. You can then use the services of a family of data access routines to exchange data with the slaves.

Listing 19.7 is a simple user mode driver that performs common operations on an I²C EEPROM from user space. The EEPROM is the same as the one discussed in Chapter 8 and has two memory banks and a slave address corresponding to each bank. The listing uses inline functions from *i2c-dev.h* to operate on device nodes associated with the banks. Get this header file from the lm-sensors package (also discussed in Chapter 8) downloadable from www.lm-sensors.org. This file contains user space equivalents for all kernel space I²C access functions listed in Table 8.1 of Chapter 8.

Listing 19.7. A User Space I²C/SMBus Driver

Code View:

```
#include <linux/i2c.h>
#include <linux/i2c-dev.h>

/* Bus addresses of the memory banks */
#define SLAVE_ADDR1  0x60
#define SLAVE_ADDR2  0x61

int main(int argc, char *argv[])
{

  /* Open the host adapter */
  if ((smbus_fp = open("/dev/i2c-0", O_RDWR)) < 0) {
    exit(1);
  }

  /* Connect to the first bank */
  if (ioctl(smbus_fp, I2C_SLAVE, EEPROM_SLAVE_ADDR1) < 0) {
    exit(1);
  }

  /* ... */
  /* Dump data from the device */
  for (reg=0; reg < length; reg++) {
    /* See i2c-dev.h from the lm-sensors package for the
       implementation of the following inline function */
    res = i2c_smbus_read_byte_data(smbus_fp, (unsigned char) reg);
    if (res < 0) {
      exit(1);
    }

    /* Dump data */
    /* ... */
  }

  /* ... */
```

```
  /* Switch to bank 2 */
  if (ioctl(smbus_fp, I2C_SLAVE, SLAVE_ADDR2) < 0) {
    exit(1);
  }

  /* Clear bank 2 */
  for (reg=0; reg < length; reg+=2){
    i2c_smbus_write_word_data(smbus_fp, (unsigned char) reg, 0x0);
  }

  /* ... */

  close(smbus_fp);
}
```

UIO

Starting with the 2.6.23 release, the kernel includes a subsystem called UIO (*Userspace IO*) that eases the implementation of some user-space drivers. UIO's intent is to allow the development of a bare-bones kernel driver for tasks such as interrupt handling, and push most of the device I/O logic to user space. UIO is especially relevant for some classes of industrial I/O cards.

Look inside *drivers/uio/* for the UIO sources. A user guide is available under *Documentation/DocBook/uio-howto.tmpl.* Exploring UIO is beyond the scope of this chapter.

Looking at the Sources

The Linux scheduler lives in *kernel/sched.c.* The SCSI generic implementation is in *drivers/scsi/sg.c*, and *drivers/usb/core/devio.c* is responsible for supporting user space USB drivers. The i2c-dev driver that enables support for user mode I^2C programming resides in *drivers/i2c/i2c-dev.c*.

Table 19.1 contains the main data structures used in this chapter, and Table 19.2 lists the functions that we used to aid user mode driver development.

Table 19.1. Summary of Data Structures

Data Structure	Location (User Space)	Description
`sched_param`	*/usr/include/bits/sched.h*	Information related to scheduling priorities.
`fb_var_screeninfo`	*/usr/include/linux/fb.h*	Used to operate on frame buffers. Contains variable screen information such as resolution and pixclock. See Chapter 11 for more details.
`sg_io_hdr_t`	*/usr/include/scsi/sg.h*	Information to manage SCSI generic devices.

`usb_dev_handle`
`usb_bus`
`usb_device`
	Header files in the libusb package.	Structures to operate on USB devices from user space.

Table 19.2. Summary of User-Space Functions

User-Space Function	Description
`sched_getparam()`	Obtains scheduling parameters associated with a given process
`sched_setscheduler()`	Sets scheduling parameters associated with a given process
`mlockall()`	Locks pages of the calling process in memory and thus avoids page faults
`ioperm()`	Controls access permissions to the first `0x3FF` I/O ports
`iopl()`	Controls access privileges to all I/O ports
`outb()/outw()/outl()`	Outputs a byte/word/long to a specified port
`inb()/inw()/inl()`	Inputs a byte/word/long from a specified port
`mmap()`	Associates a file or a device address region with a chunk of user virtual memory
`msync()`	Flushes changes made to an mmap-ed memory area
`munmap()`	Reverse of `mmap()`
`usb_init()` `usb_find_buses()`	Functions provided by the libusb library to help you operate over usbfs

User-Space Function	Description

`usb_find_devices()`
`usb_open()`
`usb_control_msg()`
`usb_close()`	

`i2c_smbus_read_byte_data()` `i2c_smbus_write_word_data()`	User-space I^2C/SMBus data access routines available as part of the lm-sensors package

Chapter 20. More Devices and Drivers

In This Chapter

- ECC Reporting 578
- Frequency Scaling 583
- Embedded Controllers 584
- ACPI 585
- ISA and MCA 587
- FireWire 588
- Intelligent Input/Output 589
- Amateur Radio 590
- Voice over IP 590
- High-Speed Interconnects 591

So far, we have devoted a full chapter to each major device driver class, but there are several subdirectories under *drivers/* that we haven't yet descended into. In this chapter let's venture inside some of them at a brisk pace.

ECC Reporting

Several memory controllers contain special silicon to measure the fidelity of stored data using *error correcting codes* (ECCs). The *Error Detection And Correction* (EDAC) driver subsystem announces occurrences of memory error events generated by ECC-aware memory controllers. Typical ECC DRAM chips have the capability to correct *single-bit errors* (SBEs) and detect *multibit errors* (MBEs). In EDAC parlance, the former errors are *correctable errors* (CEs), whereas the latter are *uncorrectable errors* (UEs).

ECC operations are transparent to the operating system. This means that if your DRAM controller supports ECC, error correction and detection occurs silently without operating system participation. EDAC's task is to report such events and allow users to fashion error handling policies (such as replace a suspect DRAM chip).

The EDAC driver subsystem consists of the following:

- A core module called *edac_mc* that provides a set of library routines.
- Separate drivers for interacting with supported memory controllers. For example, the driver module that works with the memory controller that is part of the Intel 82860 North Bridge is called *i82860_edac*.

EDAC reports errors via files in the sysfs directory, */sys/devices/system/edac/.* It also generates messages that can be gleaned from the kernel error log.

The layout of DRAM chips is specified in terms of the number of chip-selects emanating from the memory controller and the data-transfer width (channels) between the memory controller and the CPU. The number of rows in the DRAM chip array depends on the former, whereas the number of columns hinge on the latter. One of the main aims of EDAC is to point the needle of suspicion at problem DRAM chips, so the EDAC sysfs node structure is designed according to the physical chip layout: */sys/devices/system/edac/mc/mcX/csrowY/* corresponds to chip-select row Y in memory controller X. Each such directory contains details such as the number of detected CEs (`ce_count`), UEs (`ue_count`), channel location, and other attributes.

Device Example: ECC-Aware Memory Controller

Let's add EDAC support for a yet-unsupported memory controller. Assume that you're putting Linux onto a medical grade device that is an embedded x86 derivative. The North Bridge chipset (which includes the memory controller as discussed in the sidebar "The North Bridge" in Chapter 12, "Video Drivers") on your board is the Intel 855GME that is capable of ECC reporting. All DRAM banks connected to the 855GME on this system are ECC-enabled chips because this is a life-critical device. EDAC does not yet support the 855GME, so let's take a stab at implementing it.

ECC DRAM controllers have two major ECC-related registers: an error status register and an error address pointer register, as shown in Table 20.1. When an ECC error occurs, the former contains the status (whether the error is an SBE or an MBE), whereas the latter contains the physical address where the error occurred. The EDAC core periodically checks these registers and reports results to user space via sysfs. From a configuration standpoint, all devices inside the 855GME appear to be on PCIbus 0. The DRAM controller resides on device 0 of this bus. DRAM interface control registers (including the ECC-specific registers) map into the corresponding PCI configuration space. To add EDAC support for the 855GME, add hooks to read these registers, as shown in Listing 20.1. Refer back to Chapter 10, "Peripheral Component Interconnect," for explanations on PCI device driver methods and data structures.

Table 20.1. ECC-Related Registers on the DRAM Controller

ECC-Specific Registers Residing in the DRAM Controller's PCI Configuration Space	Description
I855_ERRSTS_REGISTER	The error status register, which signals occurrence of an ECC error. Shows whether the error is an SBE or an MBE.
I855_EAP_REGISTER	The error address pointer register, which contains the physical address where the most recent ECC error occurred.

Listing 20.1. An EDAC Driver for the 855GME

Code View:

```
/* Based on drivers/edac/i82860_edac.c */

#define I855_PCI_DEVICE_ID   0x3584 /* PCI Device ID of the memory
                                       controller in the 855 GME */
#define I855_ERRSTS_REGISTER 0x62   /* Error Status Register's offset
                                       in the PCI configuration space */
#define I855_EAP_REGISTER    0x98   /* Error Address Pointer Register's
                                       offset in the PCI configuration space */
struct i855_error_info {
  u16 errsts;  /* Error Type */
  u32 eap;     /* Error Location */
};

/* Get error information */
static void
i855_get_error_info(struct mem_ctl_info *mci,
                    struct i855_error_info *info)
{
  struct pci_dev *pdev;

  pdev = to_pci_dev(mci->dev);
  /* Read error type */
  pci_read_config_word(pdev, I855_ERRSTS_REGISTER, &info->errsts);
  /* Read error location */
  pci_read_config_dword(pdev, I855_EAP_REGISTER, &info->eap);
}

/* Process errors */
static int
i855_process_error_info(struct mem_ctl_info *mci,
                        struct i855_error_info *info,
                        int handle_errors)
{
  int row;

  info->eap >>= PAGE_SHIFT;
  row = edac_mc_find_csrow_by_page(mci, info->eap); /* Find culprit row */

  /* Handle using services provided by the EDAC core.
     Populate sysfs, generate error messages, and so on */
  if (is_MBE()) {          /* is_MBE() looks at I855_ERRSTS_REGISTER and checks
                              for an MBE. Implementation not shown */
```

```
    edac_mc_handle_ue(mci, info->eap, 0, row, "i855 UE");
  } else if (is_SBE()) {   /* is_SBE() looks at I855_ERRSTS_REGISTER and checks
                              for an SBE. Implementation not shown */
    edac_mc_handle_ce(mci, info->eap, 0, info->derrsyn, row, 0,
                      "i855 CE");
  }

  return 1;
}

/* This method is registered with the EDAC core from i855_probe() */
static void
i855_check(struct mem_ctl_info *mci)
{
  struct i855_error_info info;

  i855_get_error_info(mci, &info);
  i855_process_error_info(mci, &info, 1);
}

/* The PCI driver probe method, part of the pci_driver structure */
static int
i855_probe(struct pci_dev *pdev, int dev_idx)
{
  struct mem_ctl_info *mci;

  /* ... */
  pci_enable_device(pdev);

  /* Allocate control memory for this memory controller.
     The 3 arguments to edac_mc_alloc() correspond to the
     amount of requested private storage, number of chip-select
     rows, and number of channels in your memory layout */
  mci = edac_mc_alloc(0, CSROWS, CHANNELS);
  /* ... */
  mci->edac_check = i855_check; /* Supply the check method to the
                                   EDAC core */
  /* Do other memory controller initializations */
  /* ... */
  /* Register this memory controller with the EDAC core */
  edac_mc_add_mc(mci, 0);
  /* ... */
}

/* Remove method */
static void __devexit
i855_remove(struct pci_dev *pdev)
{
  struct mem_ctl_info *mci = edac_mc_find_mci_by_pdev(pdev);
  if (mci && !edac_mc_del_mc(mci)) {
    edac_mc_free(mci); /* Free memory for this controller. Reverse
                          of edac_mc_alloc() */
  }
}

/* PCI Device ID Table */
static const struct pci_device_id i855_pci_tbl[] __devinitdata = {
  {PCI_VEND_DEV(INTEL, I855_PCI_DEVICE_ID),
   PCI_ANY_ID, PCI_ANY_ID, 0, 0,},
```

```
  {0,},
};

MODULE_DEVICE_TABLE(pci, i855_pci_tbl);

/* pci_driver structure for this device.
   Re-visit Chapter 10 for a detailed explanation */
static struct pci_driver i855_driver = {
  .name     = "855",
  .probe    = i855_probe,
  .remove   = __devexit_p(i855_remove),
  .id_table = i855_pci_tbl,
};

/* Driver Initialization */
static int __init
i855_init(void)
{
  /* ... */
  pci_rc = pci_register_driver(&i855_driver);
  /* ... */
}
```

Look at *drivers/edac/** for EDAC source files and at *Documentation/drivers/edac/edac.txt* for detailed semantics of EDAC sysfs nodes.

Chapter 20. More Devices and Drivers

In This Chapter

- ECC Reporting 578
- Frequency Scaling 583
- Embedded Controllers 584
- ACPI 585
- ISA and MCA 587
- FireWire 588
- Intelligent Input/Output 589
- Amateur Radio 590
- Voice over IP 590
- High-Speed Interconnects 591

So far, we have devoted a full chapter to each major device driver class, but there are several subdirectories under *drivers/* that we haven't yet descended into. In this chapter let's venture inside some of them at a brisk pace.

ECC Reporting

Several memory controllers contain special silicon to measure the fidelity of stored data using *error correcting codes* (ECCs). The *Error Detection And Correction* (EDAC) driver subsystem announces occurrences of memory error events generated by ECC-aware memory controllers. Typical ECC DRAM chips have the capability to correct *single-bit errors* (SBEs) and detect *multibit errors* (MBEs). In EDAC parlance, the former errors are *correctable errors* (CEs), whereas the latter are *uncorrectable errors* (UEs).

ECC operations are transparent to the operating system. This means that if your DRAM controller supports ECC, error correction and detection occurs silently without operating system participation. EDAC's task is to report such events and allow users to fashion error handling policies (such as replace a suspect DRAM chip).

The EDAC driver subsystem consists of the following:

- A core module called *edac_mc* that provides a set of library routines.
- Separate drivers for interacting with supported memory controllers. For example, the driver module that works with the memory controller that is part of the Intel 82860 North Bridge is called *i82860_edac*.

EDAC reports errors via files in the sysfs directory, */sys/devices/system/edac/.* It also generates messages that can be gleaned from the kernel error log.

The layout of DRAM chips is specified in terms of the number of chip-selects emanating from the memory controller and the data-transfer width (channels) between the memory controller and the CPU. The number of rows in the DRAM chip array depends on the former, whereas the number of columns hinge on the latter. One of the main aims of EDAC is to point the needle of suspicion at problem DRAM chips, so the EDAC sysfs node structure is designed according to the physical chip layout: */sys/devices/system/edac/mc/mcX/csrowY/* corresponds to chip-select row Y in memory controller X. Each such directory contains details such as the number of detected CEs (`ce_count`), UEs (`ue_count`), channel location, and other attributes.

Device Example: ECC-Aware Memory Controller

Let's add EDAC support for a yet-unsupported memory controller. Assume that you're putting Linux onto a medical grade device that is an embedded x86 derivative. The North Bridge chipset (which includes the memory controller as discussed in the sidebar "The North Bridge" in Chapter 12, "Video Drivers") on your board is the Intel 855GME that is capable of ECC reporting. All DRAM banks connected to the 855GME on this system are ECC-enabled chips because this is a life-critical device. EDAC does not yet support the 855GME, so let's take a stab at implementing it.

ECC DRAM controllers have two major ECC-related registers: an error status register and an error address pointer register, as shown in Table 20.1. When an ECC error occurs, the former contains the status (whether the error is an SBE or an MBE), whereas the latter contains the physical address where the error occurred. The EDAC core periodically checks these registers and reports results to user space via sysfs. From a configuration standpoint, all devices inside the 855GME appear to be on PCIbus 0. The DRAM controller resides on device 0 of this bus. DRAM interface control registers (including the ECC-specific registers) map into the corresponding PCI configuration space. To add EDAC support for the 855GME, add hooks to read these registers, as shown in Listing 20.1. Refer back to Chapter 10, "Peripheral Component Interconnect," for explanations on PCI device driver methods and data structures.

Table 20.1. ECC-Related Registers on the DRAM Controller

ECC-Specific Registers Residing in the DRAM Controller's PCI Configuration Space	Description
I855_ERRSTS_REGISTER	The error status register, which signals occurrence of an ECC error. Shows whether the error is an SBE or an MBE.
I855_EAP_REGISTER	The error address pointer register, which contains the physical address where the most recent ECC error occurred.

Listing 20.1. An EDAC Driver for the 855GME

Code View:

```
/* Based on drivers/edac/i82860_edac.c */

#define I855_PCI_DEVICE_ID   0x3584 /* PCI Device ID of the memory
                                       controller in the 855 GME */
#define I855_ERRSTS_REGISTER 0x62   /* Error Status Register's offset
                                       in the PCI configuration space */
#define I855_EAP_REGISTER    0x98   /* Error Address Pointer Register's
                                       offset in the PCI configuration space */
struct i855_error_info {
  u16 errsts;  /* Error Type */
  u32 eap;     /* Error Location */
};

/* Get error information */
static void
i855_get_error_info(struct mem_ctl_info *mci,
                    struct i855_error_info *info)
{
  struct pci_dev *pdev;

  pdev = to_pci_dev(mci->dev);
  /* Read error type */
  pci_read_config_word(pdev, I855_ERRSTS_REGISTER, &info->errsts);
  /* Read error location */
  pci_read_config_dword(pdev, I855_EAP_REGISTER, &info->eap);
}

/* Process errors */
static int
i855_process_error_info(struct mem_ctl_info *mci,
                        struct i855_error_info *info,
                        int handle_errors)
{
  int row;

  info->eap >>= PAGE_SHIFT;
  row = edac_mc_find_csrow_by_page(mci, info->eap); /* Find culprit row */

  /* Handle using services provided by the EDAC core.
     Populate sysfs, generate error messages, and so on */
  if (is_MBE()) {          /* is_MBE() looks at I855_ERRSTS_REGISTER and checks
                              for an MBE. Implementation not shown */
```

```
    edac_mc_handle_ue(mci, info->eap, 0, row, "i855 UE");
  } else if (is_SBE()) {   /* is_SBE() looks at I855_ERRSTS_REGISTER and checks
                              for an SBE. Implementation not shown */
    edac_mc_handle_ce(mci, info->eap, 0, info->derrsyn, row, 0,
                      "i855 CE");
  }

  return 1;
}

/* This method is registered with the EDAC core from i855_probe() */
static void
i855_check(struct mem_ctl_info *mci)
{
  struct i855_error_info info;

  i855_get_error_info(mci, &info);
  i855_process_error_info(mci, &info, 1);
}

/* The PCI driver probe method, part of the pci_driver structure */
static int
i855_probe(struct pci_dev *pdev, int dev_idx)
{
  struct mem_ctl_info *mci;

  /* ... */
  pci_enable_device(pdev);

  /* Allocate control memory for this memory controller.
     The 3 arguments to edac_mc_alloc() correspond to the
     amount of requested private storage, number of chip-select
     rows, and number of channels in your memory layout */
  mci = edac_mc_alloc(0, CSROWS, CHANNELS);
  /* ... */
  mci->edac_check = i855_check; /* Supply the check method to the
                                   EDAC core */
  /* Do other memory controller initializations */
  /* ... */
  /* Register this memory controller with the EDAC core */
  edac_mc_add_mc(mci, 0);
  /* ... */
}

/* Remove method */
static void __devexit
i855_remove(struct pci_dev *pdev)
{
  struct mem_ctl_info *mci = edac_mc_find_mci_by_pdev(pdev);
  if (mci && !edac_mc_del_mc(mci)) {
    edac_mc_free(mci); /* Free memory for this controller. Reverse
                          of edac_mc_alloc() */
  }
}

/* PCI Device ID Table */
static const struct pci_device_id i855_pci_tbl[] __devinitdata = {
  {PCI_VEND_DEV(INTEL, I855_PCI_DEVICE_ID),
   PCI_ANY_ID, PCI_ANY_ID, 0, 0,},
```

```
  {0,},
};

MODULE_DEVICE_TABLE(pci, i855_pci_tbl);

/* pci_driver structure for this device.
   Re-visit Chapter 10 for a detailed explanation */
static struct pci_driver i855_driver = {
  .name     = "855",
  .probe    = i855_probe,
  .remove   = __devexit_p(i855_remove),
  .id_table = i855_pci_tbl,
};

/* Driver Initialization */
static int __init
i855_init(void)
{
  /* ... */
  pci_rc = pci_register_driver(&i855_driver);
  /* ... */
}
```

Look at *drivers/edac/** for EDAC source files and at *Documentation/drivers/edac/edac.txt* for detailed semantics of EDAC sysfs nodes.

Frequency Scaling

The CPU frequency (*cpufreq*) driver subsystem aids power management by scaling CPU frequencies on-the-fly. If you use a suitable scaling algorithm (called a *governor*), your device's battery can potentially last longer. Cpufreq supports several architectures such as x86, ARM, and PowerPC. To obtain cpufreq capabilities, you also need to enable a suitable processor driver (say, the Intel Enhanced SpeedStep driver if you are using a SpeedStep-enabled CPU such as Pentium M).

You can control cpufreq's behavior via files in the */sys/devices/system/cpu/cpuX/cpufreq/* directory, where *X* is the CPU number. To set maximum and minimum frequency scaling limits, write desired values to `scaling_max_freq` and `scaling_min_freq`, respectively. To see a list of supported cpufreq governors, look at the contents of `scaling_available_governors`. The kernel supports several governors:

- The *performance* governor statically sets the CPU frequency to `scaling_max_freq`.
- *Powersave* sets the CPU frequency to `scaling_min_freq`.
- *Ondemand* adjusts the frequency depending on CPU load.
- *Conservative* is a variant of ondemand where the speed change occurs smoothly in gradual steps.
- *Userspace* lets applications dictate the scaling technique. Some distributions set the governor to userspace and implement the scaling algorithm via a daemon called *cpuspeed*, which is spawned during boot.
- You may also implement your own kernel governor using the `cpufreq_register_governor()` interface.

Each supported governor is implemented as a kernel module. To see cpufreq in action, assign a governor and vary the system load:

```
bash> cd /sys/devices/system/cpu/cpu0/cpufreq
bash>cat scaling_max_freq          → Maximum frequency
1700000
bash> cat scaling_min_freq         → Minimum frequency
600000
bash> cat cpuinfo_cur_freq         → Current frequency
600000
bash> cat scaling_governor         → Scaling algorithm in use
powersave
bash> cat scaling_available_frequencies
1700000 1400000 1200000 1000000 800000 600000
bash> cat scaling_available_governors
conservative ondemand powersave userspace performance
bash> echo conservative > scaling_governor
                                   → Assign 'conservative' governor
bash> ls -lR /                     → Switch to another terminal and
                                     load your system by recursively
```

```
traversing all directories.
```

If you now monitor the running frequency by looking at */sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq*, you will see it dancing to the tune of the CPU load.

The CPU scaling code lives in the *drivers/cpufreq/* directory. Look at *Documentation/cpu-freq/** for the detailed semantics of cpufreq sysfs nodes.

Embedded Controllers

Notebook computers and their derivatives usually contain a built-in *embedded controller* (EC) to take care of various side responsibilities, including the following:

- Interfacing with the keyboard controller
- Managing thermal events
- Handling special buttons and LEDs
- Controlling system and CPU fan speeds
- Monitoring battery voltage

Most of these functions are specific to the OEM's hardware implementation. Different OEMs use different ECs; IBM/Lenovo laptops, for example, embed a Renesas H8 microcontroller to assist the main processor. The interface to access the EC, however, is standard irrespective of the make of the controller. The BIOS and the operating system operate on I/O port `0x80` to read information from the EC and I/O port `0x81` to write data to the EC. On desktops, these ports provide access to the keyboard controller rather than to a general-purpose EC.

The next section refers to an example driver that detects telemetry strength by accessing EC memory space.

ACPI

Advanced Configuration and Power Interface (ACPI) is a power-management specification that replaces earlier standards such as *Advanced Power Management* (APM). ACPI is responsible for transitioning the system between power states. It also has the task of interfacing with devices and sensors connected to the EC. Such devices are called *ACPI devices*, and memory devoted to handle them is called *ACPI space*.

As you saw elsewhere in this book, low-level code is not the place to implement policy. This was the main problem with APM, where most of the power-management policies were part of BIOS firmware. ACPI shifts policy one level up, to the operating system. Using a daemon called *acpid*, ACPI even allows policy to be pushed one more level up, to user-space configuration files. By adding rules to an *acpid* configuration file, you can decide what to do when a hotkey is pressed or when a thermal trip occurs.

Even with ACPI, low-level BIOS firmware retains the responsibility of interfacing with hardware and detecting ACPI events such as a power button press or a thermal sensor report. To perform this, the BIOS utilizes a special x86 execution mode triggered via *system management interrupts* (SMIs). The SMI execution mode is transparent to the operating system. To notify the operating system about ACPI events detected in SMI mode, the BIOS asserts a *system control interrupt* (SCI). Look at *drivers/acpi/osl.c* for the Linux ACPI code that requests the SCI IRQ.

Linux ACPI components include the following:

1. A core layer that provides ACPI essentials such as the *ACPI Machine Language* (AML) interpreter. ACPI-specific BIOS code is written in AML, a language that runs on a virtual machine implemented by the operating system's AML interpreter.

2. ACPI drivers for interfacing with standard components such as the EC (*drivers/acpi/ec.c*), buttons (*drivers/acpi/button.c*), and fan (*drivers/acpi/fan.c*). OEM-specific drivers offer support for features not supported by the standard ACPI drivers. For example, *drivers/misc/thinkpad_acpi.c*[1] is the OEM-specific driver that implements extras for IBM/Lenovo Thinkpads. On an IBM/Lenovo Thinkpad, the files under */proc/acpi/* are generated by the standard ACPI drivers, whereas those in */proc/acpi/ibm/* are produced by the OEM-specific driver. So, to get the current temperature, do this:

   [1] Prior to 2.6.22, this driver used to be *drivers/acpi/ibm_acpi.c*.

   ```
   bash> cat /proc/acpi/thermal_zone/THM0/temperature
   temperature:       39 C
   ```

   But to turn on the nightlight on top of the LCD display, get help from the OEM-specific driver:

   ```
   bash> echo on > /proc/acpi/ibm/light
   ```

3. A kernel thread *kacpid* that ACPI uses to queue work for execution.

4. Individual device drivers that use ACPI's services to respond to transitions in the system's power state. To achieve this, drivers register `suspend()` and `resume()` methods with the kernel's device model. We alluded to these methods while discussing the `platform_driver` structure in Chapter 6, "Serial Drivers," the `spi_driver` structure in Chapter 8, "The Inter-Integrated Circuit Protocol," the `pcmcia_driver` structure in Chapter 9, "PCMCIA and Compact Flash," and the `pci_driver` structure in Chapter 10.

5. User-space tools such as *acpitool*, which report the state of various ACPI devices, show thermal zone information and suspend the system to different sleep states:

```
bash> acpitool
Battery #1     : charging, 69.08%, 01:14:02
AC adapter     : on-line
Thermal zone 1 : ok, 38 C
```

6. The *acpid* daemon, which is the policy enabler for ACPI events. It listens on */proc/acpi/events* for power-management events reported by the kernel. When you press the power button or when a thermal trip occurs, the kernel ACPI driver dispatches an event to user space via */proc/acpi/events.* Acpid reads this, passes it through configuration scripts present in */etc/acpi/events/* and takes specified actions. Assume that you want to execute a specific program (*/bin/lidhandler*) when your laptop's lid button is pressed. For this, add the following to */etc/acpi/events/acpi_handler.sh*:

```
event=button/lid.*
action=/bin/lidhandler
```

   You may use cpufreq in tandem with ACPI. You can, for example, add this line inside */bin/lidhandler* to drop down the processor frequency when you shut your laptop's lid:

```
echo powersave > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

You can download the ACPI specifications from www.acpi.info.

As an exercise, consider that you have a telemetry card[2] built in to an embedded laptop derivative, and that the EC is connected to a sensor that measures telemetry strength. To access telemetry strength via */proc/acpi/* (or */sys/bus/acpi/*), update the corresponding laptop model's "extras" driver present in *drivers/misc/.* If your board is a derivative of an IBM/Lenovo Thinkpad, for example, modify *drivers/misc/thinkpad_acpi.c* accordingly. You may use the `ec_read()` and `ec_write()` kernel functions to access the location that stores telemetry strength in the EC's ACPI space.

[2] We developed a driver for an example telemetry card in Chapter 11, "Universal Serial Bus."

ISA and MCA

The *Industries Standard Architecture* (ISA) started as a bus for interfacing I/O devices with the PC but evolved into a de facto standard. ISA drivers would have merited a separate chapter several years earlier; but today, with the advent of the PCI bus, ISA has all but disappeared.

There are two main bus-specific factors that ISA device drivers have to contend with:

- ISA does not offer standard interfaces that drivers can use to detect resource information that is electrically wired or assigned by boot firmware. Implementing complex probing logic, often leveraging device-specific quirks, is an important part of ISA driver initialization. This is unlike the PCI bus, where the device driver can cleanly decipher the identity of resources such as interrupt request lines and I/O base addresses assigned by boot firmware. You learned how to do this when we discussed the PCI configuration space in Chapter 10. We also briefly looked at ISA probing in the section "ISA Network Drivers" in Chapter 15, "Network Interface Cards."

  The ISA *Plug-and-Play* (PnP) specification attempts to bring a degree of autoconfigurability to ISA, however.

- The ISA bus has a width of 24 bits, so devices can access only the low 16MB of system memory. To DMA network data from an ISA Ethernet card, for example, DMA buffers have to reside in the low 16MB range called `ZONE_DMA`. The *Extended Industry Standard Architecture* (EISA), however, widens the ISA bus to 32 bits. You can plug ISA devices into EISA slots.

Today, the LPC bus is used rather than the ISA bus to connect legacy peripherals to the CPU on PC-compatible systems. We discussed LPC devices such as Super I/O chipsets, firmware hubs, and thermal sensors in earlier chapters.

The *Micro-Channel Architecture* (MCA) bus overcomes many of the limitations of the ISA family. MCA supports bus mastering, autoconfiguration, and 32-bit bus widths. Though technologically superior to ISA, MCA didn't become as popular because of its proprietary nature.

Look at *drivers/net/tokenring/skisa.c* for a sample ISA driver for a Token Ring card. The IBM Token Ring driver, *drivers/net/tokenring/ibmtr.c*, supports ISA, PnP, and MCA form factors of IBM Token Ring hardware. The 3COM Ethernet driver, *drivers/net/3c509.c*, drives MCA, PnP, and EISA form factors of a 3COM Ethernet card. The kernel provides core routines for the use of PnP, EISA, and MCA drivers. These implementations live in *drivers/pnp/, drivers/eisa/*, and *drivers/mca/*, respectively.

FireWire

FireWire, or IEEE 1394, is a high-speed serial bus protocol invented by Apple for connecting peripheral devices to a system. It provides data rates of up to 800Mbps (IEEE 1394b). Figure 10.1 in Chapter 10 shows the connection of the FireWire controller on an x86-based laptop.

FireWire is similar to USB 2.0 in that both are external I/O buses that support high speeds and device hotplugging. FireWire, however, is a peer-to-peer protocol, unlike the master-slave USB 2.0, so two FireWire-enabled devices can exchange information without the intervention of a PC. Because of this characteristic, FireWire is popular on multimedia devices such as camcorders. As you learned in Chapter 11, the On-The-Go supplement brings peer-to-peer capability to USB 2.0, too.

FireWire on Linux is architected as follows:

- Device drivers such as *ohci1394* that interface with FireWire controllers.
- Protocol drivers for applications such as storage, video, and networking. The FireWire *Serial Bus Protocol 2* (SBP2) driver is a low-level FireWire protocol driver that lets you use your FireWire storage media as you would use a SCSI disk or a USB mass storage device. SBP2 has to be used in tandem with a high-level SCSI driver such as *sd_mod* (for disks) or *sr_mod* (for CD-ROMs). Applications such as *cdrecord* work over FireWire CD drives just as they work with USB CD drives. The *dv1394* and *video1394* protocol drivers enable you to capture video via FireWire, and the *eth1394* protocol driver lets you run TCP/IP over FireWire.
- A FireWire core that provides services to both previously mentioned.
- User-space libraries such as *libraw1394* that assist in developing FireWire-aware applications.

Look at *drivers/ieee1394/** for FireWire kernel sources and go to www.linux1394.org for detailed documentation.

Starting with the 2.6.22 release, the kernel has an alternate, slimmer FireWire stack living in the *drivers/firewire/* directory.

Intelligent Input/Output

Intelligent Input/Output (or I2O) is a standard that calls for offloading I/O activities from the main processor to an I/O coprocessor residing on an I2O adapter. I2O is largely defunct today, and the *I2O Special Interest Group* (I2O SIG) has ceased to exist. However, many operating systems, including Linux, continue support for I2O.

I2O hardware is available for technologies such as SCSI, RAID, and networking. I2O partitions the software architecture into an *OS-specific module* (OSM) running on the main processor and a *hardware-specific module* (HDM) executing on the I2O adapter. HDMs are OS-agnostic and can be reused across operating systems, so the OSMs are rendered simpler.

Linux supports I2O in the form of an I2O core, drivers for I2O adapters, and various OSMs. Look at the Linux I2O home page at http://i2o.shadowconnect.com and the sources in *drivers/message/i2o/* for more details.

Amateur Radio

Amateur (ham) radio is a packet radio technology used for round-the-world communication by hobbyists. It's also often used to respond to calamities such as floods and cyclones. To use amateur radio on Linux, you need the following:

- A low-level modem driver to access your radio. Modem drivers for several amateur radio devices are present in *drivers/net/hamradio/*.

- One or more packet protocols such as AX.25, Rose, and Netrom. The AX.25 protocol is an adaptation of the X.25 protocol for amateur radio. Look at the Linux Amateur Radio AX.25 HOWTO for an explanation of the protocol, the *net/ax25/* directory for the sources, and http://hams.sourceforge.net for user-space utilities and libraries that operate over AX.25. Rose (*net/rose/*) and Netrom (*net/netrom/*) are network protocols that use AX.25 as the data link layer. You can write Linux socket applications that run over AX.25, Rose, and Netrom using the `AF_AX25`, `AF_ROSE`, and `AF_NETROM` protocol families, respectively.

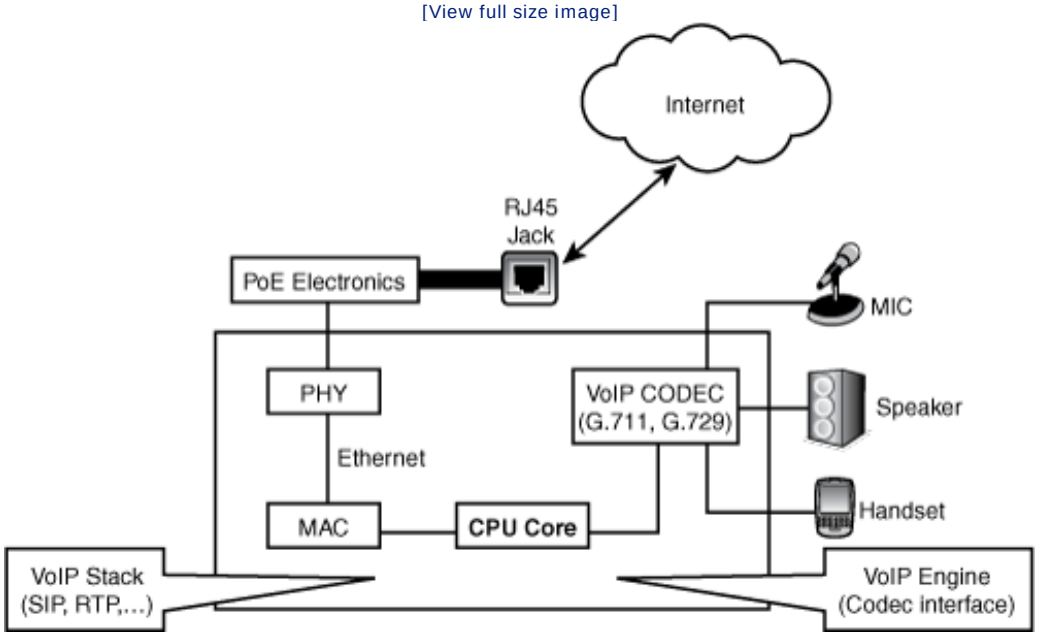
Voice over IP

Voice over Internet Protocol (VoIP) is a technology that uses the Internet to carry voice traffic. VoIP lets you make voice-quality telephone calls at cheap rates. There are several PCI-, PC Card-, and USB-based VoIP solutions available for the PC environment. Device drivers for several of these cards are available on Linux. Not many are integrated into the mainline kernel, however. The *drivers/telephony/* directory contains drivers for a few VoIP devices and a registration API that future drivers can use.

With the increasing popularity of Linux in the embedded telecom space, there are several Linux IP telephones in the market today. Figure 20.1 shows a VoIP-enabled device having a hardware voice codec that implements standards such as G.711 and G.729 for encoding and decoding voice streams. The device draws power using a technology called *Power over Ethernet* (PoE) that transmits power along with the Ethernet cable. A device driver communicates with the VoIP hardware.

Figure 20.1. A VoIP phone.

[View full size image]



VoIP drivers work in tandem with transport protocols such as *Real Time Transport Protocol* (RTP) and call control signaling stacks such as *Session Initiation Protocol* (SIP) and H.323. On top of these protocols sit various IP telephony applications.

Solutions that implement VoIP codecs in software are also popular in the embedded space. They usually reside in user space and interact with the following:

- Kernel audio drivers using OSS or ALSA APIs
- Kernel network drivers using the socket API

SoCs oriented toward the *Video-and-Voice over IP* (V2IP) market usually contain hardware support for video codecs such as H.264. If you are putting Linux onto a V2IP phone, you need to implement drivers to interface with such codecs, too.

High-Speed Interconnects

High-speed interconnecting technologies such as InfiniBand, RapidIO, Hyper-Transport, and 10 Gigabit Ethernet are not common in the PC or low-end embedded environments. You are more likely to find them on clusters, blade servers, gaming systems, switches, or high-speed routers. Networking technologies such as Fibre Channel and *Internet SCSI* (iSCSI) can be found in enterprise environments served by *storage-area networks* (SANs).

Let's peek at the driver subsystems for some of these technologies.

InfiniBand

InfiniBand is a high-speed serial bus standard originally intended to replace PCI. PCI Express, however, has become the accepted future of system buses. Today, InfiniBand technology is commonly used in blade server designs to provide a high-performance storage and networking fabric. InfiniBand supports *Remote DMA* (RDMA), which allows data to be DMA-ed from the memory of one computer system to another.

The Linux InfiniBand subsystem includes core support for InfiniBand, device drivers for host channel adapters, and an implementation of IP over InfiniBand. Look inside *drivers/infiniband/* for the Linux InfiniBand subsystem and at *Documentation/infiniband/** for related documentation.

RapidIO

RapidIO is another high-speed serial bus technology, which is used for connecting boards via a back plane. It supports speeds of the order of 10Gbps. An example processor that supports RapidIO is the power-based MPC8540 from Freescale, targeted at embedded devices such as network routers and switches.

The Linux RapidIO subsystem provides a core set of routines that can be used to drive devices on the RapidIO bus. There are two ways to communicate over a -RapidIO interconnect:

1. Short, out-of-band messages using *doorbells.* Doorbell services provided by the RapidIO core are `rio_request_inb_dbell()`, `rio_release_inb_dbell()`, `rio_request_outb_dbell()`, and `rio_release_outb_dbell()`.

2. High-bandwidth data delivery using *mailboxes.* Mailbox services provided by the RapidIO core are `rio_request_inb_mbox()`, `rio_release_inb_mbox()`, `rio_request_outb_mbox()`, and `rio_release_outb_mbox()`.

Take a look inside *drivers/rapidio/* for the sources.

Fibre Channel

Fibre Channel is a modern high-speed serial bus protocol used to talk with storage systems. Fibre Channel interface cards have fiber-optic ports to talk to storage devices on SANs. Fibre Channel is compatible with SCSI, so a Fibre Channel device driver is essentially a SCSI driver with extras to handle fiber channels.

Linux supports a Fibre Channel core and device drivers to handle Fibre Channel hardware. Look inside *drivers/fc4/* for the sources.

iSCSI

iSCSI is another SAN technology. It allows the transport of SCSI packets over TCP/IP networks. With iSCSI, a remote block device appears to your system as local storage. The remote system owning the storage is called an *iSCSI target*, and local systems using the storage are called *iSCSI initiators*.

Linux supports iSCSI via a kernel driver, *drivers/scsi/iscsi_tcp.c*, and a user-space daemon called *iscsid.* The home page of the Linux-iSCSI project is at http://linux-iscsi.sourceforge.net.

Chapter 21. Debugging Device Drivers

In This Chapter

- Kernel Debuggers 596
- Kernel Probes 609
- Kexec and Kdump 620
- Profiling 629
- Tracing 634
- Linux Test Project 638
- User Mode Linux 638
- Diagnostic Tools 638
- Kernel Hacking Config Options 639
- Test Equipment 640

Now that we have learned how to implement diverse classes of device drivers, let's take a step back and explore some debugging techniques. Investing time in logic design and software engineering before code development and staring hard at the code after development can minimize or even eliminate bugs. But because that is easier said than done, and because we are all humans, developers need debugging tools. In this chapter, let's look at a variety of methods to debug kernel code.

Reliability, Availability, Serviceability

Many systems, especially mission critical ones, have a need for reliability, availability, and serviceability (RAS). The Linux RAS effort has resulted in the development of several powerful tools. Exercisers such as the Linux Test Project (LTP) measure the reliability and robustness of your kernel port. CPU hotplugging and the Linux High Availability (HA) project can be seen in the context of availability. Kernel debuggers, Kprobes, Kdump, EDAC, and the Linux Trace Toolkit (LTT) come under the ambit of serviceability. The line dividing these classifications is sometimes thin; you can use any or a combination of these methods to suit your debugging needs. For example, output from a kernel profiler such as *OProfile* can be used either to search for potential code bottlenecks (reliability) or to debug a field problem (serviceability).

Kernel Debuggers

The Linux kernel has no built-in debugger support. Whether to include a debugger as part of the stock kernel is an oft-debated point in kernel mailing lists. The instruction level *Kernel Debugger* (kdb) and the source-level *Kernel GNU Debugger* (kgdb) are the two main Linux kernel debuggers. As of today, whether you use kdb or kgdb, you need to download relevant patches and apply them to your kernel sources. Even if you want to stay away from the hassle of patching your kernel sources with debugger support, you can glean information about kernel panics and peek at kernel variables via the plain *GNU Debugger* (gdb). JTAG debuggers use hardware-assisted debugging and are powerful, but expensive.

Kernel debuggers make kernel internals more transparent. You can single-step through instructions, disassemble instructions, display and modify kernel variables, and look at stack traces. In this chapter, let's learn the basics of kernel debuggers with the help of some examples.

Entering a Debugger

You can enter a kernel debugger in multiple ways. One way is to pass command-line arguments that ask the kernel to enter the debugger during boot. Another way is via software or hardware *breakpoint*s. A breakpoint is an address where you want execution stopped and control transferred to the debugger. A software breakpoint replaces the instruction at that address with something else that causes an exception. You may set software breakpoints either using debugger commands or by inserting them into your code. For x86-based systems, you can set a software breakpoint in your kernel source code as follows:

```
asm(" int $3");
```

Alternatively, you can invoke the `BREAKPOINT` macro, which translates to the appropriate architecture-dependent instruction.

You may use hardware breakpoints in place of software breakpoints if the instruction where you need to stop is in flash memory, where it cannot be replaced by the debugger. A hardware breakpoint needs processor support. The corresponding address has to be added to a debug register. You can only have as many hardware

breakpoints as the number of debug registers supported by the processor.

You may also ask a debugger to set a *watchpoint* on a variable. The debugger stops execution whenever an instruction modifies data at the watchpoint address.

Yet another common method to enter a debugger is by hitting an attention key, but there are instances when this won't work. If your code is sitting in a tight loop after disabling interrupts, the kernel will not get a chance to process the attention key and enter the debugger. For example, you can't enter the debugger via an attention key if your code does something like this:

```
unsigned long flags;

local_irq_save(flags);
while (1) continue;
local_irq_restore(flags);
```

When control is transferred to the debugger, you can start your analysis using various debugger commands.

Kernel Debugger (kdb)

Kdb is an instruction-level debugger used for debugging kernel code and device drivers. Before you can use it, you need to patch your kernel sources with kdb support and recompile the kernel. (Refer to the section "Downloads" for information on downloading kdb patches.) The main advantage of kdb is that it's easy to set up, because you don't need an additional machine to do the debugging (unlike kgdb). The main disadvantage is that you need to correlate your sources with disassembled code (again, unlike kgdb).

Let's wet our toes in kdb with the help of an example. Here's the crime scene: You have modified a kernel serial driver to work with your x86-based hardware. But the driver isn't working, and you would like kdb to help nab the culprit.

Let's start our search for fingerprints by setting a breakpoint at the serial driver `open()` entry point. Remember, because kdb is not a source-level debugger, you have to open your sources and try to match the instructions with your C code. Let's list the source snippet in question:

***drivers/serial/myserial.c*:**

```
static int rs_open(struct tty_struct *tty, struct file *filp)
{
  struct async_struct *info;

  /* ... */
  retval = get_async_struct(line, &info);
  if (retval) return(retval);
  tty->driver_data = info;
  /* Point A */

  /* ... */
}
```

Press the Pause key and enter kdb. Let's find out how the disassembled `rs_open()` looks. As usual, all debug sessions shown in this chapter attach explanations using the → symbol.

```
Entering kdb (current=0xc03f6000, pid 0) on processor 0 due to
Keyboard Entry
```

```
kdb> id rs_open            → Disassemble rs_open
0xc01cce00 rs_open:        sub $0x1c, %esp
0xc01cce03 rs_open+0x03:   mov $fffffed, %ecx
...
0xc01cce4b rs_open+0x4b:   call 0xc01ccca0, get_async_struct
...
0xc01cce56 rs_open+0x56:   mov 0xc(%esp,1), %eax
0xc01cce5a rs_open+0x5a:   mov %eax, 0x9a4(%ebx)
...
```

Point A in the source code is a good place to attach a breakpoint because you can peek at both the `tty` structure and the `info` structure to see what's going on.

Looking side by side at the source and the disassembly, `rs_open+0x5a` corresponds to Point A. Note that correlation is easier if the kernel is compiled without optimization flags.

Set a breakpoint at `rs_open+0x5a` (which is address `0xc01cce5a`) and continue execution by exiting the debugger:

```
kbd> bp rs_open+0x5a   → Set breakpoint
kbd> go                → Continue execution
```

Now you need to get the kernel to call `rs_open()` to hit the breakpoint. To trigger this, execute an appropriate user-space program. In this case, echo some characters to the corresponding serial port (*/dev/ttySX*):

```
bash> echo "kerala monsoons" > /dev/ttySX
```

This results in the invocation of `rs_open()`. The breakpoint gets hit, and kdb assumes control:

```
Entering kdb on processor 0 due to Breakpoint @ 0xc01cce5a
kdb>
```

Let's now find out the contents of the `info` structure. If you look at the disassembly, one instruction before the breakpoint (`rs_open+0x56`), you will see that the `EAX` register contains the address of the `info` structure. Let's look at the register contents:

```
kbd> r                 → Dump register contents
eax = 0xcf1ae680 ebx = 0xce03b000 ecx = 0x00000000
...
```

So, `0xcf1ae680` is the address of the `info` structure. Dump its contents using the `md` command:

```
kbd> md 0xcf1ae680     → Memory dump
0xcf1ae680 00005301 0000ABC 00000000 10000400
...
```

To make sense of this dump, let's look at the corresponding structure definition. `info` is defined as `struct async_struct` in *include/linux/serialP.h* as follows:

```
struct async_struct {
  int            magic;  /* Magic Number */
  unsigned long  port;   /* I/O Port */
  int            hub6;
  /* ... */
};
```

If you match the dump with the definition, `0x5301` is the magic number and `0xABC` is the I/O port. Well, isn't this interesting! `0xABC` doesn't look like a valid port. If you have done enough serial port debugging, you will know that the I/O port base addresses and IRQs are configured in *include/asm-x86/serial.h* for x86-based hardware. Change the port definition to the correct value, recompile the kernel, and continue your testing!

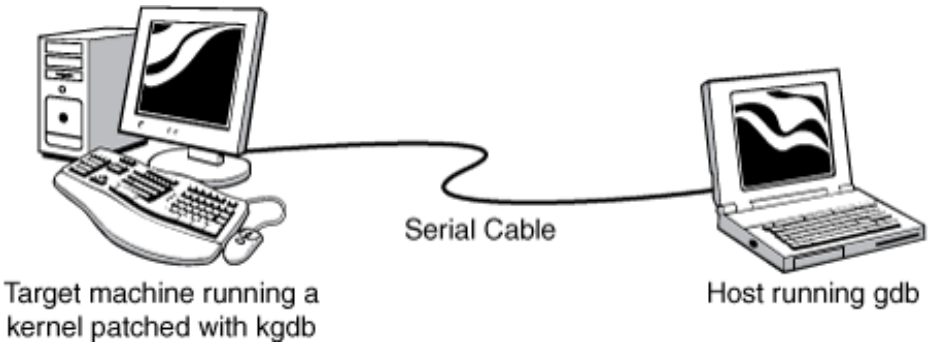
Kernel GNU Debugger (kgdb)

Kgdb is a source-level debugger. It is easier to use than kdb because you don't have to spend time correlating assembly code with your sources. However it's more difficult to set up because an additional machine is needed to front-end the debugging.

You have to use gdb in tandem with kgdb to step through kernel code. gdb runs on the host machine, while the kgdb-patched kernel (refer to the "Downloads" section for information on downloading kgdb patches) runs on the target hardware. The host and the target are connected via a serial null-modem cable, as shown in Figure 21.1.[1]

[1] You can also launch kgdb debug sessions over Ethernet.

Figure 21.1. Kgdb setup.



You have to inform the kernel about the identity and baud rate of the serial port via command-line arguments. Depending on the bootloader used, add the following kernel arguments to either *syslinux.cfg*, *lilo.conf*, or *grub.conf*:

```
kgdbwait kgdb8250=X,115200
```

`kgdbwait` asks the kernel to wait until a connection is established with the host-side gdb, `X` is the serial port connected to the host, and `115200` is the baud rate used for communication.

Now configure the same baud rate on the host side:

```
bash> stty speed 115200 < /dev/ttySX
```

If your host computer is a laptop that does not have a serial port, you may use a USB-to-serial converter for the debug session. In that case, instead of */dev/ttySX*, use the */dev/ttyUSBX* node created by the usbserial driver. Figure 6.4 of Chapter 6, "Serial Drivers," illustrates this scenario.

Let's learn kgdb basics using the example of a buggy kernel module. Modules are easier to debug because the entire kernel need not be recompiled after making code changes, but remember to compile your module with the `-g` option to generate symbolic information. Because modules are dynamically loaded, the debugger needs to be informed about the symbolic information that the module contains. Listing 21.1 contains a buggy `trojan_function()`. Assume that it's defined in *drivers/char/my_module.c*.

Listing 21.1. Buggy Function

```
char buffer;

int
trojan_function()
{
  int *my_variable = 0xAB, i;

  /* ... */
  Point A:
  i = *my_variable; /* Kernel Panic: my_variable points
                       to bad memory */
  return(i);
}
```

Insert *my_module.ko* on the target and look inside */sys/module/my_module/sections/* to decipher ELF (*Executable and Linking Format*) section addresses.[2] The `.text` section in ELF files contains code, `.data` contains initialized variables, `.rodata` contains initialized read-only variables such as strings, and `.bss` contains variables that are not initialized during startup. The addresses of these sections are available in the form of the files *.text*, *.data*, *.rodata*, and *.bss* in */sys/module/my_module/sections/* if you enable `CONFIG_KALLSYMS` during kernel configuration. To obtain the code section address, for instance, do this:

[2] If you are still using a 2.4 kernel, get the section addresses using the `–m` option to insmod instead:

```
bash> insmod my_module.o –m
Using /lib/modules/2.x.y/kernel/drivers/char/my_module.o
Sections:       Size      Address   Align
.this           00000060  e091a000  2**2
.text           00001ec0  e091a060  2**4
...
.rodata         0000004c  e091d1fc  2**2
.data           00000048  e091d260  2**5
.bss            000000e4  e091d2c0  2**5
...
```

```
bash> cat /sys/module/my_module/sections/.text
0xe091a060
```

More module load information is available from */proc/modules* and */proc/kallsyms*.

After you have the section addresses, invoke gdb on the host-side machine:

```
bash> gdb vmlinux              → vmlinux is the uncompressed kernel

(gdb) target remote /dev/ttySX → Connect to the target
```

Because you passed `kgdbwait` as a kernel command-line argument, gdb gets control when the kernel boots on the target. Now inform gdb about the preceding section addresses using the `add-symbol-file` command:

```
(gdb) add-symbol-file drivers/char/mymodule.ko 0xe091a060
      -s .rodata 0xe091d1fc -s .data 0xe091d260 -s .bss 0xe091d2c0

add symbol table from file "drivers/char/my_module.ko" at
        .text_addr = 0xe091a060
        .rodata_addr = 0xe091d1fc
        .data_addr = 0xe091d260
        .bss_addr = 0xe091d2c0
(y or n) y
Reading symbols from drivers/char/mymodule.ko ...done.
```

To debug the kernel panic, let's set a breakpoint at `trojan_function()`:

```
(gdb) b trojan_function        → Set breakpoint
(gdb) c                        → Continue execution
```

When kgdb hits the breakpoint, let's look at the stack trace, single-step until Point A, and display the value of `my_variable`:

```
(gdb) bt                       → Back (stack) trace
#0 trojan_function () at my_module.c :124
#1 0xe091a108 in my_parent_function (my_var1=438, my_var2=0xe091d288)
..

(gdb) step
(gdb) step                     → Continue to single-step up to
                                  Point A
(gdb) p my_variable
$0 = 0
```

There is an obvious bug here. `my_variable` points to `NULL` because `trojan_function()` forgot to allocate memory for it. Let's just allocate the memory using kgdb, circumvent the kernel crash, and continue testing:

```
(gdb) p &buffer                → Print address of buffer
$1 = 0xe091a100 ""
(gdb) set my_variable=0xe091a100 → my_variable = &buffer
```

```
(gdb) c                          → Continue your testing
```

Kgdb ports are available for several architectures such as x86, ARM, and PowerPC. When you use kgdb to debug a target embedded device (instead of the PC shown in Figure 21.1), the gdb front-end that you run on your host system needs to be compiled to work with your target platform. For example, to debug a device driver developed for an ARM-based embedded device from your x86-based host development system, you have to use the appropriately generated gdb, often named arm-linux-gdb. The exact name depends on the distribution you use.

GNU Debugger (gdb)

As mentioned earlier, you can use plain gdb to gather some kernel debug information. However, you can't step through kernel code, set breakpoints, or modify kernel variables. Let's use gdb to debug the kernel panic caused by the buggy function in Listing 21.1, but assume this time that `trojan_function()` is compiled as part of the kernel and not as a module, because you can't easily peek inside modules using gdb.

This is part of the "oops" message generated when `trojan_function()` is executed:

```
Unable to handle kernel NULL pointer dereference at
virtual address 000000ab
 ...
 eax: f7571de0   ebx: ffffe000   ecx: f6c78000   edx: f98df870
 ...
 Stack: c019d731 00000000
        ...
        bffffbe8 c0108fab
Call Trace:     [<c019d731>] [<c013b8ac>] [<c0108fab>]
...
```

Copy this cryptic "oops" message to *oops.txt* and use the *ksymoops* utility to obtain more verbose output. You might need to hand-copy the message if the system is hung:

```
bash> ksymoops oops.txt
Code;  c019d710 <trojan_function+0/10>
00000000 <_EIP>:
Code;  c019d710 <trojan_function+0/10>  <=====
   0:   a1 ab 00 00 00            mov   0xab,%eax   <=====
Code;  c019d715 <trojan_function+5/10>
   5:   c3                        ret
```

2.6 kernels emit "oops" output that can be used as is without the need of decoding using *ksymoops* if you enable `CONFIG_KALLSYMS` during kernel configuration.

Looking at the preceding dump, the "oops" has occurred inside `trojan_function()`. Let's use gdb to obtain more information. In the following invocation, *vmlinux* is the uncompressed kernel image, and */proc/kcore* is the kernel address space:

```
bash> gdb vmlinux /proc/kcore
```

```
(gdb) p xtime          → Test the waters by printing a kernel variable
$0 = 1113173755
```

Repeated access to the same variable will not yield refreshed values due to gdb's cached access. You can force a fresh access by rereading the core file using gdb's `core-file` command. Let's now look at the disassembly of `trojan_function()`:

```
(gdb) x/2i trojan_function        → Disassemble trojan_function
0xc019d710 <trojan_function>:    mov 0xab, %eax
0xc019d715 <trojan_function+5>:  ret
```

`trojan_function()` looks laconic when seen in assembly due to compiler optimizations. It's effectively copying the contents of address `0xab` to the `EAX` register, which holds the return value from functions on x86-based systems. But `0xab` does not look like a valid kernel address! Fix the bug by allocating valid memory space to `my_variable`, recompile, and continue your testing.

JTAG Debuggers

JTAG debuggers use hardware-assist to debug code. You need specialized monitor hardware[3] and a front-end user interface (some JTAG debuggers use gdb as the front-end) to step through code. JTAG can also be used for purposes other than debugging, such as burning code onto on-board flash memory, as discussed in Chapter 18, "Embedding Linux." JTAG connectors are common on development boards but are usually not part of production units.

[3] Some JTAG debuggers work with several processor architectures if you suitably replace the probe that connects the debugger to the target board.

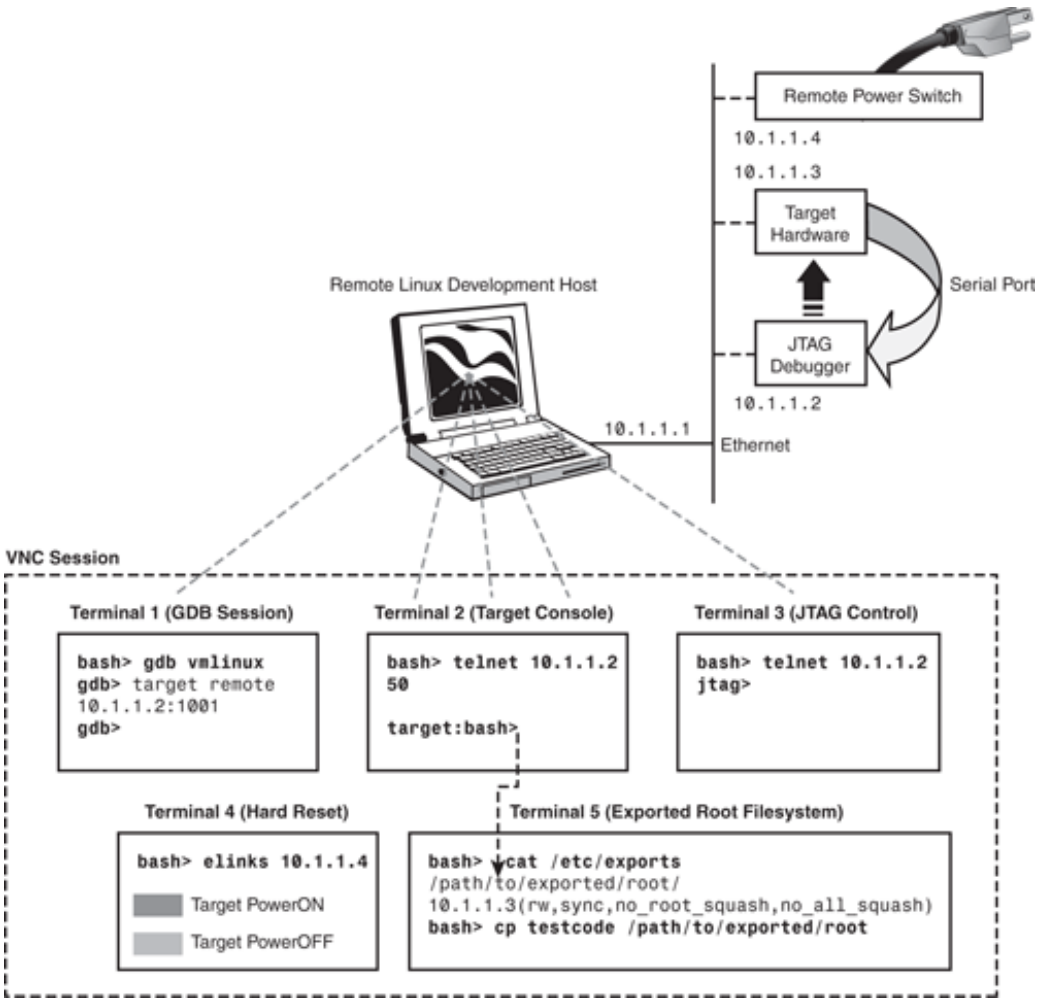
JTAG debuggers usually connect to target hardware via serial port, USB, or Ethernet. With Ethernet, you can remotely access the JTAG debugger, and hence the target board, even if the board itself does not possess a network interface.

Figure 21.2 shows a JTAG-based remote debugging session in action. The JTAG debugger used in this scenario supports a gdb front end. The development host and the JTAG hardware are connected to an Ethernet LAN. The debug serial port on the target hardware is connected to the serial port on the JTAG box. Figure 21.2 achieves remote debugging on the Linux development host using five terminal sessions. Terminal 1 runs gdb, which connects to the JTAG box over the network using telnet:

```
(gdb) target remote 10.1.1.2:1001 → 10.1.1.2 is the IP address of
                                    the JTAG hardware. 1001 is the
                                    JTAG TCP port that listens to
                                    incoming gdb connections
```

Figure 21.2. An example JTAG-based remote debug setup.

[View full size image]



To debug boot portions of the kernel, for example, set a gdb breakpoint at `start_kernel()`. (You can find its address from *System.map*, which is generated in the root of your source tree when you build the kernel.)

Terminal 2 attaches a serial console to the target. A telnet client running on Terminal 2 connects to a prespecified TCP port on the JTAG box, which is configured (using Terminal 3) to tunnel data arriving via its serial port:

```
bash> telnet 10.1.1.2 50          → 10.1.1.2 is the IP address of
                                    the JTAG hardware. 50 is the
                                    JTAG TCP port that tunnels data
                                    arriving via its serial port
```

This is equivalent to running an emulator such as *minicom* after directly connecting the target's debug serial port to the host (instead of to the JTAG box, as shown in Figure 21.2), but that'll constrain the host to be physically adjacent to the target.

Terminal 3 telnets to the JTAG box and offers debugger-specific semantics. You can use it for example, to do the following:

- Pull a JTAG definition script over TFTP from the host and execute it during JTAG boot. A JTAG definition

script usually initializes the processor, clock registers, chip select registers, and memory banks. After this is done, the JTAG hardware is ready to download code on to the target and execute it. The JTAG manufacturer usually provides definition files for all supported platforms, so you are likely to have a close starting point for your board.

- Download your bootloader, kernel, or stand-alone code from the host over TFTP, to flash memory or RAM on the target. File formats such as ELF and binary are usually supported by JTAG debuggers.
- Single-step code, set breakpoints, examine registers, and dump memory regions.
- Reset the target.

JTAG debugging can be flaky at times, so if you are debugging remotely, it might be a good idea to power the target via a remote power control switch, as shown in Figure 21.2. That way, you can hard-reset the target from the host using a web browser, as shown in Terminal 4. You may also choose to power the JTAG hardware via a remote power switch. That will let you test run a bootloader directly from flash without the intervention of JTAG and its definition files.

If the target board possesses a network interface, it can mount its root filesystem over NFS from the development host. (See the section "NFS-Mounted Root" in Chapter 18 for details on doing this.) Terminal 5 on the host operates locally on the exported root filesystem.[4]

[4] You may have more such terminals depending on your debug scenario. If you are using an oscilloscope that has remote display capabilities, for example, you may operate it via a web browser on another terminal.

If your team is scattered geographically, run Terminals 1 through 5 within an environment such as *Virtual Network Computing* (VNC). If VNC is not already part of your distribution, download it from www.realvnc.com. With such a setup, you can debug the electrons on your remote board from the comfort of your home! Some JTAG vendors provide a sophisticated integrated development environment[5] that encompasses all the functionalities previously detailed, so you don't need to manage VNC terminal sessions if you're using one of those.

[5] While JTAG hardware is independent of the target operating system, the front-end interface is likely to have OS dependencies.

During hardware bring up, when you are porting your bootloader or other stand-alone code to the target, it's a good idea to first generate an ELF image and debug it from RAM before running it from flash. Remember, however, to eliminate bootloader initializations that duplicate the ones performed by the JTAG definition script.

A key advantage of JTAG debuggers is that you can use a single tool to debug the different pieces that constitute your firmware solution. So, you can use the same debugger to debug the BIOS, bootloader, base kernel, device driver modules, as well as user-space applications, at source level.

Downloads

You may download kdb patches for the x86 and IA64 architectures from http://oss.sgi.com/projects/kdb. Each supported kernel version needs two patches: a common one and an architecture-dependent one.

The home page for the kgdb project is http://kgdb.sourceforge.net. The website also has documentation on configuring and using kgdb.

If your Linux distribution does not already contain gdb, you can obtain it from www.gnu.org/software/gdb/gdb.html.

Chapter 21. Debugging Device Drivers

In This Chapter

- Kernel Debuggers 596
- Kernel Probes 609
- Kexec and Kdump 620
- Profiling 629
- Tracing 634
- Linux Test Project 638
- User Mode Linux 638
- Diagnostic Tools 638
- Kernel Hacking Config Options 639
- Test Equipment 640

Now that we have learned how to implement diverse classes of device drivers, let's take a step back and explore some debugging techniques. Investing time in logic design and software engineering before code development and staring hard at the code after development can minimize or even eliminate bugs. But because that is easier said than done, and because we are all humans, developers need debugging tools. In this chapter, let's look at a variety of methods to debug kernel code.

Reliability, Availability, Serviceability

Many systems, especially mission critical ones, have a need for reliability, availability, and serviceability (RAS). The Linux RAS effort has resulted in the development of several powerful tools. Exercisers such as the Linux Test Project (LTP) measure the reliability and robustness of your kernel port. CPU hotplugging and the Linux High Availability (HA) project can be seen in the context of availability. Kernel debuggers, Kprobes, Kdump, EDAC, and the Linux Trace Toolkit (LTT) come under the ambit of serviceability. The line dividing these classifications is sometimes thin; you can use any or a combination of these methods to suit your debugging needs. For example, output from a kernel profiler such as *OProfile* can be used either to search for potential code bottlenecks (reliability) or to debug a field problem (serviceability).

Kernel Debuggers

The Linux kernel has no built-in debugger support. Whether to include a debugger as part of the stock kernel is an oft-debated point in kernel mailing lists. The instruction level *Kernel Debugger* (kdb) and the source-level *Kernel GNU Debugger* (kgdb) are the two main Linux kernel debuggers. As of today, whether you use kdb or kgdb, you need to download relevant patches and apply them to your kernel sources. Even if you want to stay away from the hassle of patching your kernel sources with debugger support, you can glean information about kernel panics and peek at kernel variables via the plain *GNU Debugger* (gdb). JTAG debuggers use hardware-assisted debugging and are powerful, but expensive.

Kernel debuggers make kernel internals more transparent. You can single-step through instructions, disassemble instructions, display and modify kernel variables, and look at stack traces. In this chapter, let's learn the basics of kernel debuggers with the help of some examples.

Entering a Debugger

You can enter a kernel debugger in multiple ways. One way is to pass command-line arguments that ask the kernel to enter the debugger during boot. Another way is via software or hardware *breakpoint*s. A breakpoint is an address where you want execution stopped and control transferred to the debugger. A software breakpoint replaces the instruction at that address with something else that causes an exception. You may set software breakpoints either using debugger commands or by inserting them into your code. For x86-based systems, you can set a software breakpoint in your kernel source code as follows:

```
asm(" int $3");
```

Alternatively, you can invoke the `BREAKPOINT` macro, which translates to the appropriate architecture-dependent instruction.

You may use hardware breakpoints in place of software breakpoints if the instruction where you need to stop is in flash memory, where it cannot be replaced by the debugger. A hardware breakpoint needs processor support. The corresponding address has to be added to a debug register. You can only have as many hardware

breakpoints as the number of debug registers supported by the processor.

You may also ask a debugger to set a *watchpoint* on a variable. The debugger stops execution whenever an instruction modifies data at the watchpoint address.

Yet another common method to enter a debugger is by hitting an attention key, but there are instances when this won't work. If your code is sitting in a tight loop after disabling interrupts, the kernel will not get a chance to process the attention key and enter the debugger. For example, you can't enter the debugger via an attention key if your code does something like this:

```
unsigned long flags;

local_irq_save(flags);
while (1) continue;
local_irq_restore(flags);
```

When control is transferred to the debugger, you can start your analysis using various debugger commands.

Kernel Debugger (kdb)

Kdb is an instruction-level debugger used for debugging kernel code and device drivers. Before you can use it, you need to patch your kernel sources with kdb support and recompile the kernel. (Refer to the section "Downloads" for information on downloading kdb patches.) The main advantage of kdb is that it's easy to set up, because you don't need an additional machine to do the debugging (unlike kgdb). The main disadvantage is that you need to correlate your sources with disassembled code (again, unlike kgdb).

Let's wet our toes in kdb with the help of an example. Here's the crime scene: You have modified a kernel serial driver to work with your x86-based hardware. But the driver isn't working, and you would like kdb to help nab the culprit.

Let's start our search for fingerprints by setting a breakpoint at the serial driver `open()` entry point. Remember, because kdb is not a source-level debugger, you have to open your sources and try to match the instructions with your C code. Let's list the source snippet in question:

***drivers/serial/myserial.c*:**

```
static int rs_open(struct tty_struct *tty, struct file *filp)
{
  struct async_struct *info;

  /* ... */
  retval = get_async_struct(line, &info);
  if (retval) return(retval);
  tty->driver_data = info;
  /* Point A */

  /* ... */
}
```

Press the Pause key and enter kdb. Let's find out how the disassembled `rs_open()` looks. As usual, all debug sessions shown in this chapter attach explanations using the → symbol.

```
Entering kdb (current=0xc03f6000, pid 0) on processor 0 due to
Keyboard Entry
```

```
kdb> id rs_open            → Disassemble rs_open
0xc01cce00 rs_open:        sub $0x1c, %esp
0xc01cce03 rs_open+0x03:   mov $fffffed, %ecx
...
0xc01cce4b rs_open+0x4b:   call 0xc01ccca0, get_async_struct
...
0xc01cce56 rs_open+0x56:   mov 0xc(%esp,1), %eax
0xc01cce5a rs_open+0x5a:   mov %eax, 0x9a4(%ebx)
...
```

Point A in the source code is a good place to attach a breakpoint because you can peek at both the `tty` structure and the `info` structure to see what's going on.

Looking side by side at the source and the disassembly, `rs_open+0x5a` corresponds to Point A. Note that correlation is easier if the kernel is compiled without optimization flags.

Set a breakpoint at `rs_open+0x5a` (which is address `0xc01cce5a`) and continue execution by exiting the debugger:

```
kbd> bp rs_open+0x5a   → Set breakpoint
kbd> go                → Continue execution
```

Now you need to get the kernel to call `rs_open()` to hit the breakpoint. To trigger this, execute an appropriate user-space program. In this case, echo some characters to the corresponding serial port (*/dev/ttySX*):

```
bash> echo "kerala monsoons" > /dev/ttySX
```

This results in the invocation of `rs_open()`. The breakpoint gets hit, and kdb assumes control:

```
Entering kdb on processor 0 due to Breakpoint @ 0xc01cce5a
kdb>
```

Let's now find out the contents of the `info` structure. If you look at the disassembly, one instruction before the breakpoint (`rs_open+0x56`), you will see that the `EAX` register contains the address of the `info` structure. Let's look at the register contents:

```
kbd> r                 → Dump register contents
eax = 0xcf1ae680 ebx = 0xce03b000 ecx = 0x00000000
...
```

So, `0xcf1ae680` is the address of the `info` structure. Dump its contents using the `md` command:

```
kbd> md 0xcf1ae680     → Memory dump
0xcf1ae680 00005301 0000ABC 00000000 10000400
...
```

To make sense of this dump, let's look at the corresponding structure definition. `info` is defined as `struct async_struct` in *include/linux/serialP.h* as follows:

```
struct async_struct {
  int            magic;  /* Magic Number */
  unsigned long  port;   /* I/O Port */
  int            hub6;
  /* ... */
};
```

If you match the dump with the definition, `0x5301` is the magic number and `0xABC` is the I/O port. Well, isn't this interesting! `0xABC` doesn't look like a valid port. If you have done enough serial port debugging, you will know that the I/O port base addresses and IRQs are configured in *include/asm-x86/serial.h* for x86-based hardware. Change the port definition to the correct value, recompile the kernel, and continue your testing!

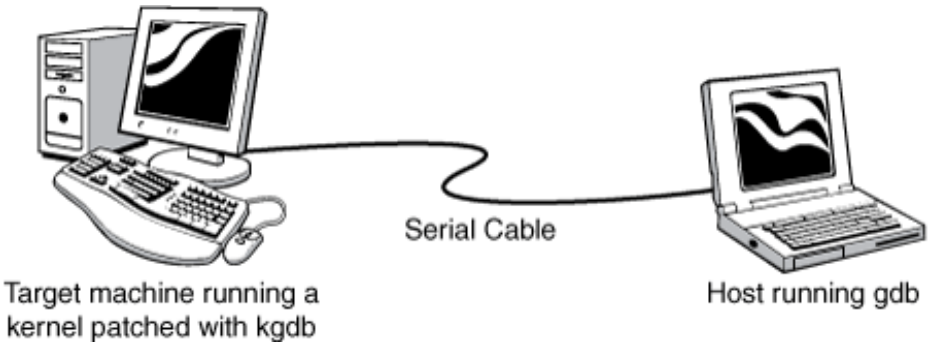
Kernel GNU Debugger (kgdb)

Kgdb is a source-level debugger. It is easier to use than kdb because you don't have to spend time correlating assembly code with your sources. However it's more difficult to set up because an additional machine is needed to front-end the debugging.

You have to use gdb in tandem with kgdb to step through kernel code. gdb runs on the host machine, while the kgdb-patched kernel (refer to the "Downloads" section for information on downloading kgdb patches) runs on the target hardware. The host and the target are connected via a serial null-modem cable, as shown in Figure 21.1.[1]

[1] You can also launch kgdb debug sessions over Ethernet.

Figure 21.1. Kgdb setup.



You have to inform the kernel about the identity and baud rate of the serial port via command-line arguments. Depending on the bootloader used, add the following kernel arguments to either *syslinux.cfg*, *lilo.conf*, or *grub.conf*:

```
kgdbwait kgdb8250=X,115200
```

`kgdbwait` asks the kernel to wait until a connection is established with the host-side gdb, `X` is the serial port connected to the host, and `115200` is the baud rate used for communication.

Now configure the same baud rate on the host side:

```
bash> stty speed 115200 < /dev/ttySX
```

If your host computer is a laptop that does not have a serial port, you may use a USB-to-serial converter for the debug session. In that case, instead of */dev/ttySX*, use the */dev/ttyUSBX* node created by the usbserial driver. Figure 6.4 of Chapter 6, "Serial Drivers," illustrates this scenario.

Let's learn kgdb basics using the example of a buggy kernel module. Modules are easier to debug because the entire kernel need not be recompiled after making code changes, but remember to compile your module with the `-g` option to generate symbolic information. Because modules are dynamically loaded, the debugger needs to be informed about the symbolic information that the module contains. Listing 21.1 contains a buggy `trojan_function()`. Assume that it's defined in *drivers/char/my_module.c*.

Listing 21.1. Buggy Function

```
char buffer;

int
trojan_function()
{
  int *my_variable = 0xAB, i;

  /* ... */
  Point A:
  i = *my_variable; /* Kernel Panic: my_variable points
                       to bad memory */
  return(i);
}
```

Insert *my_module.ko* on the target and look inside */sys/module/my_module/sections/* to decipher ELF (*Executable and Linking Format*) section addresses.[2] The `.text` section in ELF files contains code, `.data` contains initialized variables, `.rodata` contains initialized read-only variables such as strings, and `.bss` contains variables that are not initialized during startup. The addresses of these sections are available in the form of the files *.text*, *.data*, *.rodata*, and *.bss* in */sys/module/my_module/sections/* if you enable `CONFIG_KALLSYMS` during kernel configuration. To obtain the code section address, for instance, do this:

[2] If you are still using a 2.4 kernel, get the section addresses using the `–m` option to insmod instead:

```
bash> insmod my_module.o –m
Using /lib/modules/2.x.y/kernel/drivers/char/my_module.o
Sections:       Size      Address   Align
.this           00000060  e091a000  2**2
.text           00001ec0  e091a060  2**4
...
.rodata         0000004c  e091d1fc  2**2
.data           00000048  e091d260  2**5
.bss            000000e4  e091d2c0  2**5
...
```

```
bash> cat /sys/module/my_module/sections/.text
0xe091a060
```

More module load information is available from */proc/modules* and */proc/kallsyms*.

After you have the section addresses, invoke gdb on the host-side machine:

```
bash> gdb vmlinux              → vmlinux is the uncompressed kernel

(gdb) target remote /dev/ttySX → Connect to the target
```

Because you passed `kgdbwait` as a kernel command-line argument, gdb gets control when the kernel boots on the target. Now inform gdb about the preceding section addresses using the `add-symbol-file` command:

```
(gdb) add-symbol-file drivers/char/mymodule.ko 0xe091a060
      -s .rodata 0xe091d1fc -s .data 0xe091d260 -s .bss 0xe091d2c0

add symbol table from file "drivers/char/my_module.ko" at
        .text_addr = 0xe091a060
        .rodata_addr = 0xe091d1fc
        .data_addr = 0xe091d260
        .bss_addr = 0xe091d2c0
(y or n) y
Reading symbols from drivers/char/mymodule.ko ...done.
```

To debug the kernel panic, let's set a breakpoint at `trojan_function()`:

```
(gdb) b trojan_function        → Set breakpoint
(gdb) c                        → Continue execution
```

When kgdb hits the breakpoint, let's look at the stack trace, single-step until Point A, and display the value of `my_variable`:

```
(gdb) bt                       → Back (stack) trace
#0 trojan_function () at my_module.c :124
#1 0xe091a108 in my_parent_function (my_var1=438, my_var2=0xe091d288)
..

(gdb) step
(gdb) step                     → Continue to single-step up to
                                  Point A
(gdb) p my_variable
$0 = 0
```

There is an obvious bug here. `my_variable` points to `NULL` because `trojan_function()` forgot to allocate memory for it. Let's just allocate the memory using kgdb, circumvent the kernel crash, and continue testing:

```
(gdb) p &buffer                → Print address of buffer
$1 = 0xe091a100 ""
(gdb) set my_variable=0xe091a100 → my_variable = &buffer
```

```
(gdb) c                          → Continue your testing
```

Kgdb ports are available for several architectures such as x86, ARM, and PowerPC. When you use kgdb to debug a target embedded device (instead of the PC shown in Figure 21.1), the gdb front-end that you run on your host system needs to be compiled to work with your target platform. For example, to debug a device driver developed for an ARM-based embedded device from your x86-based host development system, you have to use the appropriately generated gdb, often named arm-linux-gdb. The exact name depends on the distribution you use.

GNU Debugger (gdb)

As mentioned earlier, you can use plain gdb to gather some kernel debug information. However, you can't step through kernel code, set breakpoints, or modify kernel variables. Let's use gdb to debug the kernel panic caused by the buggy function in Listing 21.1, but assume this time that `trojan_function()` is compiled as part of the kernel and not as a module, because you can't easily peek inside modules using gdb.

This is part of the "oops" message generated when `trojan_function()` is executed:

```
Unable to handle kernel NULL pointer dereference at
virtual address 000000ab
 ...
 eax: f7571de0   ebx: ffffe000   ecx: f6c78000   edx: f98df870
 ...
 Stack: c019d731 00000000
        ...
        bffffbe8 c0108fab
Call Trace:     [<c019d731>] [<c013b8ac>] [<c0108fab>]
...
```

Copy this cryptic "oops" message to *oops.txt* and use the *ksymoops* utility to obtain more verbose output. You might need to hand-copy the message if the system is hung:

```
bash> ksymoops oops.txt
Code;  c019d710 <trojan_function+0/10>
00000000 <_EIP>:
Code;  c019d710 <trojan_function+0/10>  <=====
   0:   a1 ab 00 00 00            mov   0xab,%eax   <=====
Code;  c019d715 <trojan_function+5/10>
   5:   c3                        ret
```

2.6 kernels emit "oops" output that can be used as is without the need of decoding using *ksymoops* if you enable `CONFIG_KALLSYMS` during kernel configuration.

Looking at the preceding dump, the "oops" has occurred inside `trojan_function()`. Let's use gdb to obtain more information. In the following invocation, *vmlinux* is the uncompressed kernel image, and */proc/kcore* is the kernel address space:

```
bash> gdb vmlinux /proc/kcore
```

```
(gdb) p xtime          → Test the waters by printing a kernel variable
$0 = 1113173755
```

Repeated access to the same variable will not yield refreshed values due to gdb's cached access. You can force a fresh access by rereading the core file using gdb's `core-file` command. Let's now look at the disassembly of `trojan_function()`:

```
(gdb) x/2i trojan_function        → Disassemble trojan_function
0xc019d710 <trojan_function>:    mov 0xab, %eax
0xc019d715 <trojan_function+5>:  ret
```

`trojan_function()` looks laconic when seen in assembly due to compiler optimizations. It's effectively copying the contents of address `0xab` to the `EAX` register, which holds the return value from functions on x86-based systems. But `0xab` does not look like a valid kernel address! Fix the bug by allocating valid memory space to `my_variable`, recompile, and continue your testing.

JTAG Debuggers

JTAG debuggers use hardware-assist to debug code. You need specialized monitor hardware[3] and a front-end user interface (some JTAG debuggers use gdb as the front-end) to step through code. JTAG can also be used for purposes other than debugging, such as burning code onto on-board flash memory, as discussed in Chapter 18, "Embedding Linux." JTAG connectors are common on development boards but are usually not part of production units.

[3] Some JTAG debuggers work with several processor architectures if you suitably replace the probe that connects the debugger to the target board.

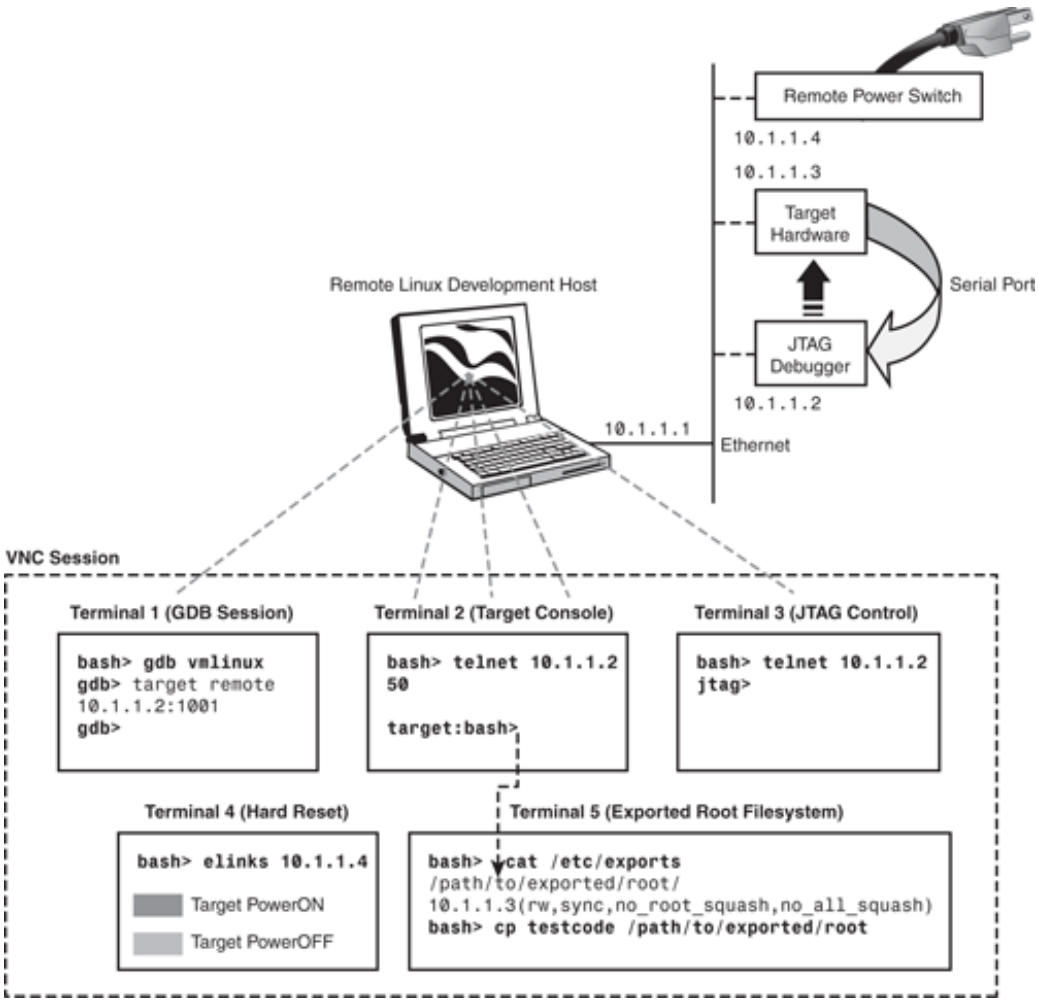
JTAG debuggers usually connect to target hardware via serial port, USB, or Ethernet. With Ethernet, you can remotely access the JTAG debugger, and hence the target board, even if the board itself does not possess a network interface.

Figure 21.2 shows a JTAG-based remote debugging session in action. The JTAG debugger used in this scenario supports a gdb front end. The development host and the JTAG hardware are connected to an Ethernet LAN. The debug serial port on the target hardware is connected to the serial port on the JTAG box. Figure 21.2 achieves remote debugging on the Linux development host using five terminal sessions. Terminal 1 runs gdb, which connects to the JTAG box over the network using telnet:

```
(gdb) target remote 10.1.1.2:1001 → 10.1.1.2 is the IP address of
                                    the JTAG hardware. 1001 is the
                                    JTAG TCP port that listens to
                                    incoming gdb connections
```

Figure 21.2. An example JTAG-based remote debug setup.

[View full size image]



To debug boot portions of the kernel, for example, set a gdb breakpoint at `start_kernel()`. (You can find its address from *System.map*, which is generated in the root of your source tree when you build the kernel.)

Terminal 2 attaches a serial console to the target. A telnet client running on Terminal 2 connects to a prespecified TCP port on the JTAG box, which is configured (using Terminal 3) to tunnel data arriving via its serial port:

```
bash> telnet 10.1.1.2 50          → 10.1.1.2 is the IP address of
                                    the JTAG hardware. 50 is the
                                    JTAG TCP port that tunnels data
                                    arriving via its serial port
```

This is equivalent to running an emulator such as *minicom* after directly connecting the target's debug serial port to the host (instead of to the JTAG box, as shown in Figure 21.2), but that'll constrain the host to be physically adjacent to the target.

Terminal 3 telnets to the JTAG box and offers debugger-specific semantics. You can use it for example, to do the following:

- Pull a JTAG definition script over TFTP from the host and execute it during JTAG boot. A JTAG definition

script usually initializes the processor, clock registers, chip select registers, and memory banks. After this is done, the JTAG hardware is ready to download code on to the target and execute it. The JTAG manufacturer usually provides definition files for all supported platforms, so you are likely to have a close starting point for your board.

- Download your bootloader, kernel, or stand-alone code from the host over TFTP, to flash memory or RAM on the target. File formats such as ELF and binary are usually supported by JTAG debuggers.
- Single-step code, set breakpoints, examine registers, and dump memory regions.
- Reset the target.

JTAG debugging can be flaky at times, so if you are debugging remotely, it might be a good idea to power the target via a remote power control switch, as shown in Figure 21.2. That way, you can hard-reset the target from the host using a web browser, as shown in Terminal 4. You may also choose to power the JTAG hardware via a remote power switch. That will let you test run a bootloader directly from flash without the intervention of JTAG and its definition files.

If the target board possesses a network interface, it can mount its root filesystem over NFS from the development host. (See the section "NFS-Mounted Root" in Chapter 18 for details on doing this.) Terminal 5 on the host operates locally on the exported root filesystem.[4]

[4] You may have more such terminals depending on your debug scenario. If you are using an oscilloscope that has remote display capabilities, for example, you may operate it via a web browser on another terminal.

If your team is scattered geographically, run Terminals 1 through 5 within an environment such as *Virtual Network Computing* (VNC). If VNC is not already part of your distribution, download it from www.realvnc.com. With such a setup, you can debug the electrons on your remote board from the comfort of your home! Some JTAG vendors provide a sophisticated integrated development environment[5] that encompasses all the functionalities previously detailed, so you don't need to manage VNC terminal sessions if you're using one of those.

[5] While JTAG hardware is independent of the target operating system, the front-end interface is likely to have OS dependencies.

During hardware bring up, when you are porting your bootloader or other stand-alone code to the target, it's a good idea to first generate an ELF image and debug it from RAM before running it from flash. Remember, however, to eliminate bootloader initializations that duplicate the ones performed by the JTAG definition script.

A key advantage of JTAG debuggers is that you can use a single tool to debug the different pieces that constitute your firmware solution. So, you can use the same debugger to debug the BIOS, bootloader, base kernel, device driver modules, as well as user-space applications, at source level.

Downloads

You may download kdb patches for the x86 and IA64 architectures from http://oss.sgi.com/projects/kdb. Each supported kernel version needs two patches: a common one and an architecture-dependent one.

The home page for the kgdb project is http://kgdb.sourceforge.net. The website also has documentation on configuring and using kgdb.

If your Linux distribution does not already contain gdb, you can obtain it from www.gnu.org/software/gdb/gdb.html.

Kernel Probes

Kernel probes can intrude into a kernel function and extract debug information or apply a medicated patch. It's a useful addition to your debugging repertoire for investigating inexplicable behavior at a customer site, especially when you don't have the luxury of rebooting the system. Linux supports a generic form of kernel probes called *Kprobes* and two specialized variants, *Jprobes* and *return probes*.

Kprobes

Kprobes can save you the trouble of building and booting a debug kernel by providing capabilities to dynamically dump kernel data structures or insert code into a running kernel. You can, for example, add a few `printk`s on-the-fly inside the scheduler without recompiling the kernel. You can even patch a bug on a Mars rover without rebooting it.

To insert a kprobe inside a kernel function, follow these steps:

1. Turn on `CONFIG_KPROBES` (*Instrumentation Support* ⟶ *Kprobes*) in the kernel configuration menu.

2. Implement a kernel module that registers a kprobe at the instruction of interest. You need to register a *pre-handler* that Kprobes will run just before executing the probed instruction and a *post-handler* that Kprobes will run after executing the probed instruction. You may also supply a *fault-handler* that will run if a fault is detected while executing the pre- or post-handlers (because you don't want to "oops" due to a debugging bug!).

When a kprobe is registered, it saves the probed instruction and replaces it with an instruction that generates a breakpoint (`int 0x03` on x86-based systems). When the breakpoint is hit, the kernel generates a *die* notification. (We discussed notifier chains in Chapter 3, "Kernel Facilities.") Kprobes inserts itself into the die notifier chain, so it gets notified about the breakpoint hit.

When notified, Kprobes executes the registered pre-handler. Next, it steps through a copy of the probed instruction. It executes a copy instead of swapping the probed instruction with the breakpoint instruction for reasons of SMP consistency. Finally, it runs the post-handler. The pre- and post-handler windows are the hooks offered to the Kprobes user to inject debug code. The handlers can be registered and unregistered on-the-fly, so serviceability is not merely static at compile time but programmable during runtime.

Let's learn to use Kprobes with the help of an example. Consider the code snippet in Listing 21.2, which is a kernel thread that adds `npages` number of pages to the free memory pool, whenever a `SIGUSR1` signal is delivered to it. Most of the logic has been scissored out of the listing because it's not relevant. Assume that you are at a customer site to debug a problem reported with this code. You notice bad things whenever `npages` crosses `10`, so you want to apply a runtime patch that limits it to `10`.

Listing 21.2. Problem Code (mydrv.c)

Code View:
```
int npages=0;
EXPORT_SYMBOL(npages);

static int memwalkd(void *unused)
{
  long curr_pfn = (64*1024*1024 >> PAGE_SHIFT);
  struct page *curr_page;
  /* ... */

  daemonize("memwalkd"); /* kernel thread */

  sigfillset(&current->blocked);
  allow_signal(SIGUSR1);

  for (;;) {
    /* Dequeue a signal if it's pending */
    if (signal_pending(current)) {
      sig = dequeue_signal(current, &current->blocked, &info);
      /* Point A */
      /* Free npages pages when SIGUSR1 is received */
      if (sig == SIGUSR1) {
      /* Point B */
        /* Problem manifests when npages crosses 10 in the following
           loop. Let's apply run time medication here via Kprobes */
        for (i=0; i < npages; i++, curr_pfn++) {
          /* ... */
        }
      }
      /* ... */
  }
  /* ... */
}
```

Listing 21.3. Registering Kprobe Handlers

Code View:
```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>
#include <linux/sched.h>

extern int npages; /* Defined in Listing 21.2 */

/* Per-probe structure */
static struct kprobe bandaid;

/* Pre Handler: Invoked before running probed instruction */
int bandaid_pre(struct kprobe *p, struct pt_regs *regs)
{
  if (npages > 10) npages = 10;
  return 0;
}
```

```
/* Post Handler: Invoked after running probed instruction */
void bandaid_post(struct kprobe *p, struct pt_regs *regs,
                  unsigned long flags)
{
  /* Nothing to do */
}
/* Fault Handler: Invoked if the pre/post-handlers
   encounter a fault */
int bandaid_fault(struct kprobe *p, struct pt_regs *regs,
                  int trapnr)
{
  return 0;
}

int init_module(void)
{
  int retval;

  /* Fill the kprobe structure */
  bandaid.pre_handler   = bandaid_pre;
  bandaid.post_handler  = bandaid_post;
  bandaid.fault_handler = bandaid_fault;

  /* Arrive at the target address as explained */
  bandaid.addr = (kprobe_opcode_t*)
                 kallsyms_lookup_name("memwalkd") + 0xaa;

  if (!bandaid.addr) {
    printk("Bad Probe Point\n");
    return -1;
  }

  /* Register the kprobe */
  if ((retval = register_kprobe(&bandaid)) < 0) {
    printk("register_kprobe error, return value=%d\n",
           retval);
    return -1;
  }
  return 0;
}

void module_cleanup(void)
{
  unregister_kprobe(&bandaid);
}

MODULE_LICENSE("GPL"); /* You can't link the Kprobes API
                          unless your user module is GPL'ed */
```

Listing 21.3 uses Kprobes to insert a patch at `kallsyms_lookup_name("memwalkd") + 0xaa`, which limits `npages` to `10`. To figure out how to arrive at this probe address, take another look at Listing 21.2. You want the patch to be inserted at Point B. To calculate the kernel address at Point B, disassemble the contents of *mydrv.ko* using *objdump*:

Code View:

```
bash> objdump -D mydrv.ko

mydrv.ko:     file format elf32-i386

Disassembly of section .text:

00000000 <memwalkd>:
   0:    55                          push   %ebp
   1:    bd 00 40 00 00              mov    $0x4000,%ebp
   6:    57                          push   %edi
   7:    56                          push   %esi
   8:    53                          push   %ebx
   9:    bb 00 f0 ff ff              mov    $0xffff000,%ebx
   e:    81 ec 90 00 00 00           sub    $0x90,%esp
  ...
  ...
  7a:    83 f8 0a                    cmp    $0xa,%eax     --> Point A
  7d:    74 2b                       je     aa <memwalkd+0xaa>
  7f:    83 f8 09                    cmp    $0x9,%eax
  82:    75 cc                       jne    50 <memwalkd+0x50>
  ...
  a9:    c3 ret
  aa:    a1 00 00 00 00              mov    0x0,%eax      --> Point B
  af:    85 c0                       test   %eax,%eax
  b1:    0f 8e b5 00 00 00           jle    16c <memwalkd+0x16c>
  b7:    81 fd 7b f6 00 00           cmp    $0xf67b,%ebp
  ...
  fa:    a1 00 00 00 00              mov    0x0,%eax
```

> You have to use an architecture-specific objdump if you're cross-compiling for a different processor platform. You will need something like arm-linux-objdump if you're disassembling a binary cross-compiled for an ARM-based target device. Pass the -S option to objdump to mix source code with the disassembled output:
>
> ```
> bash> arm-linux-objdump –d –S mydrv.ko
> ```

If you try and match the C code in Listing 21.2 with its disassembled dump above, you can associate Point A and Point B with the shown kernel addresses. `kallsyms_lookup_name()`[6] locates the address of `memwalkd()`, and `0xaa` is the offset where Point B resides, so apply the kprobe at `kallsyms_lookup_name("memwalkd") + 0xaa`.

[6] You have to enable `CONFIG_KALLSYMS` during kernel configuration to obtain the services of this function.

After you register the kprobe, `memwalkd()` equivalently looks like this:

```
static int memwalkd(void *unused)
{
  /* ...*/
  for (;;) {
    /* ... */
    /* Point A */
    /* Free npages pages when SIGUSR1 is received */
    if (sig == SIGUSR1) {
    /* Point B */
      if (npages > 10) npages = 10; /* The medicated patch! */

      for (i=0; i < npages; i++, curr_pfn++) {
        /* ... */
      }
    }
    /* ... */
  }
  /* ... */
}
```

Whenever `npages` is assigned a value greater than `10`, the kprobed patch pulls it back to `10`, thus stepping around the problem.

In the next two sections, let's look at a couple of helper facilities that make it easier to use Kprobes during function entry and exit.

Jprobes

A *jprobe* is a specialized kprobe. It eases the work of adding a probe when the point of investigation is at the entry to a kernel function. The jprobe handler has the same prototype as the probed function. It's invoked with the same argument list as the probed function, so you can easily access the function arguments from the jprobe handler. If you use Kprobes rather than Jprobes, imagine the hassles your probe handler needs to undergo, wading through the dark alleys of the function stack to extract function arguments! And this code that delves into the stack to elicit argument values has to be heavily function-specific, not to mention being architecture-dependent and unportable.

To learn how to use Jprobes, let's revert to an example. Assume that you're debugging a network device driver (that is built as part of the kernel) by looking at the `printk()` messages it's generating. The driver is emitting crucial values in octal (base 8), but to your horror, the driver writer has introduced a typo in the print format string by coding `%O` rather than `%o`. So, all you can see are messages such as this:

```
Number of Free Receive buffers = %O.
```

Jprobes to the rescue. You can fix this in a few seconds, without recompiling or rebooting the kernel. First, take a look at `printk()` defined in *kernel/printk.c*:

```
asmlinkage int printk(const char *fmt, ...)
{
  va_list args;
  int r;

  va_start(args, fmt);
  r = vprintk(fmt, args);
  va_end(args);
```

```
  return r;
}
```

Let's add a simple jprobe at the entry to `printk()` and transform every %O into %o. Listing 21.4 does this job. Note that the jprobe handler needs to have the same prototype as `printk()`. Both functions are marked with the `asmlinkage` tag that asks them to expect arguments from the stack, rather than from CPU registers.

Listing 21.4. Registering Jprobe Handlers

Code View:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

/* Jprobe the entrance to printk */
asmlinkage int
jprintk(const char *fmt, ...)
{
   for (; *fmt; ++fmt) {
     if ((*fmt=='%')&&(*(fmt+1) == 'O')) *(char *)(fmt+1) = 'o';
   }
   jprobe_return();
   return 0;
}

/* Per-probe structure */
static struct jprobe jprobe_eg = {
  .entry = (kprobe_opcode_t *) jprintk
};

int
init_module(void)
{
  int retval;

  jprobe_eg.kp.addr = (kprobe_opcode_t*)
                      kallsyms_lookup_name("printk");

  if (!jprobe_eg.kp.addr) {
    printk("Bad probe point\n");
    return -1;
  }

  /* Register the Jprobe */
  if ((retval = register_jprobe(&jprobe_eg) < 0)) {
    printk("register_jprobe error, return value=%d\n",
            retval);
    return -1;
  }
  printk("Jprobe registered.\n");
  return 0;
}

void
module_cleanup(void)
{
```

```
  unregister_jprobe(&jprobe_eg);
}

MODULE_LICENSE("GPL");
```

When Listing 21.4 invokes `register_jprobes()` to register the jprobe, a kprobe is inserted at the beginning of `printk()`. When this probe is hit, Kprobes replace the saved return address with that of the registered jprobe handler, `jprintk()`. It then copies a portion of the stack and returns, thus passing control to `jprintk()` with `printk()`'s argument list. When `jprintk()` calls `jprobe_return()`, the original call state is restored, and `printk()` continues to execute normally.

When you insert this jprobe user module, the network driver no longer emits useless messages announcing `%O` buffers, rather it prints saner information such as this:

```
Number of Free Receive buffers = 12.
```

Return Probes

A *return probe* (or a *kretprobe* in Kprobes terminology) is another specialized Kprobes helper. It eases the work of inserting a kprobe when you need to probe a function's return point. If you use vanilla Kprobes to investigate return points, you might need to register them at multiple places because a function can return via multiple code paths. However, if you use return probes, you need to insert only one kretprobe, rather than register, say, 20 Kprobes to cover a function's 20 return paths.

The function `tty_open()` defined in *drivers/char/tty_io.c* has seven return paths. The successful path returns `0`, and others return error values such as `–ENXIO` and `-ENODEV`. A single kretprobe is sufficient to alert you about failures, irrespective of the associated code path. Listing 21.5 implements this kretprobe.

Listing 21.5. Registering Return Probe Handlers

Code View:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

/* kretprobe at exit from tty_open() */
static int
kret_tty_open(struct kretprobe_instance *kreti,
              struct pt_regs *regs)
{
  /* The EAX register contains the function return value
     on x86 systems */
  if ((int) regs->eax) {
    /* tty_open() failed. Announce the return code */
    printk("tty_open returned %d\n", (int)regs->eax);
  }
  return 0;
}

/* Per-probe structure */
static struct kretprobe kretprobe_eg = {
  .handler = (kretprobe_handler_t)kret_tty_open
};
```

```
int
init_module(void)
{
  int retval;

  kretprobe_eg.kp.addr = (kprobe_opcode_t*)
                         kallsyms_lookup_name("tty_open");

  if (!kretprobe_eg.kp.addr) {
    printk("Bad Probe Point\n");
    return -1;
  }

  /* Register the kretprobe */
  if ((retval = register_kretprobe(&kretprobe_eg) < 0)) {
    printk("register_kretprobe error, return value=%d\n",
            retval);
    return -1;
  }
  printk("kretprobe registered.\n");
  return 0;
}

void module_cleanup(void)
{
  unregister_kretprobe(&kretprobe_eg);
}

MODULE_LICENSE("GPL");
```

When Listing 21.5 invokes `register_kretprobes()`, a kprobe is internally inserted at the beginning of `tty_open()`. When this probe gets hit, this internal kprobe handler replaces the function return address with that of a special routine (called a *trampoline* in Kprobes terminology). Look at *arch/your-arch/kernel/kprobes.c* for the implementation of the trampoline.

When `tty_open()` returns via any of its seven return paths, control returns to the trampoline instead of the caller function. The trampoline invokes the kretprobe handler, `kret_tty_open()` registered by Listing 21.5, which prints the return value if `tty_open()` has not returned successfully.

Limitations

Kprobes has its limitations. Some of them are obvious. You won't, for example, see desired results if you insert a kprobe inside an inline function. And, of course, you can't probe Kprobes code.

Kprobes are more useful for applying probes inside the base kernel. If the subject code is part of a dynamically loadable module, you might as well rewrite and recompile your module rather than write and compile a new module to "kprobe" it. However, you might still want to use Kprobes if bringing down the module is not acceptable.

There are less-obvious limitations, too. Optimizations are done at compile time, whereas Kprobes are inserted during runtime. So, the effect of inserting instructions via Kprobes is not equivalent to adding code in the original source files. For example, the buggy code snippet

```
volatile int *integerp = 0xFF;
```

```
int integerd = *integerp;
```

is reduced by the compiler to

```
mov 0xff, %eax
```

So, you can't easily use Kprobes if you want to sneak in between those two lines of C code, allocate a word of memory, point `integerp` to the allocated word, and circumvent a kernel crash.

SystemTap (http://sourceware.org/systemtap/) is a diagnostic tool that eases the use of Kprobes.

Looking at the Sources

The Kprobes implementation consists of a generic portion defined in *kernel/kprobes.c* (and *include/linux/kprobes.h*) and an architecture-dependent part that lives in *arch/your-arch/kernel/kprobes.c* (and *include/asm-your-arch/kprobes.h*).

Peek inside *Documentation/kprobes.txt* for further information about Kprobes, Jprobes, and Kretprobes.

Kexec and Kdump

Now that you have learned how to use Kprobes, let's continue and look at more facets of Linux RAS. *Kexec* and *kdump* are serviceability features introduced in the 2.6 kernel.

Kexec uses the image overlay philosophy of the UNIX `exec()` system call to spawn a new kernel over a running kernel without the overhead of boot firmware. This can save several seconds of reboot time because boot firmware spends cycles walking buses and recognizing devices. The less the reboot latency, the less the system downtime; so, this was one of the main motivations for developing kexec. However, kexec's most popular user is kdump. Capturing a dump after a kernel crash is inherently unreliable because kernel code that accesses the dump device might be in an unstable state. Kdump circumvents this problem by collecting the dump after booting into a healthy kernel via kexec.

Kexec

Before you can kexec a kernel, you need to do some preparations:

1. Compile and boot into a kernel that has kexec support. For this, turn on `CONFIG_KEXEC` (*Processor Type and Features* → *Kexec System Call*) in the kernel configuration menu. This kernel is called the first kernel or the running kernel.

2. Prepare the kernel that is to be kexec-ed. This second kernel can be the same as the first kernel.

3. Download the *kexec-tools* package source tar ball from www.kernel.org/pub/linux/kernel/people/horms/kexec-tools/kexec-tools-testing.tar.gz. Build and produce the user-space tool called *kexec*.

The kexec tool built in Step 3 is invoked in two stages. The first stage loads the second kernel image into the buffers of the running kernel, while the second stage actually overlays the running kernel:

1. Load the second (overlay) kernel using the kexec command:

   ```
   bash> kexec -l /path/to/kernelsources/arch/x86/boot/bzImage --
   append="root=/dev/hdaX" --initrd=/boot/myinitrd.img
   ```

   bzImage is the second kernel, *hdaX* is the root device, and *myinitrd.img* is the initial root filesystem. The kernel implementation of this stage is mostly architecture-independent. At the heart of this stage is the `sys_kexec()` system call. The kexec command loads the new kernel image into the running kernel's buffers using the services of this system call.

2. Boot into the second kernel:

   ```
   bash> kexec -e
   ...                    → Kernel boot up messages
   ```

   Kexec abruptly starts the new kernel without gracefully halting the operating system. To shut down prior to reboot, invoke kexec from the bottom of the *halt* script (usually */etc/rc.d/rc0.d/S01halt*) and invoke halt instead.

   The implementation of the second stage is architecture-dependent. The crux of this stage is a `reboot_code_buffer` that contains assembly code to put the new kernel in place to boot.

Kexec bypasses the initial kernel code that invokes the services of boot firmware and directly jumps to the protected mode entry point (for x86 processors). An important challenge to implement kexec is the interaction that happens between the kernel and the boot firmware (BIOS on x86-based systems, Openfirmware on POWER-based machines, and so on). On x86 systems, information such as the e820 memory map passed to the kernel by the BIOS (see Appendix B, "Linux and the BIOS") needs to be supplied to the kexec-ed kernel, too.

Kexec with Kdump

The kexec invocation semantics is somewhat special when it's used in tandem with kdump. In this case, kexec is required to automatically boot a new kernel when it encounters a kernel panic. If the running kernel crashes, the new kernel (called the capture kernel) is booted to reliably collect the dump. A typical call syntax is this:

```
bash> kexec -p /path/to/capture-kernel-sources/vmlinux
            --args-linux --append="root=/dev/hdaX irqpoll"
            --initrd=/boot/myinitrd.img
```

The `-p` option asks kexec to trigger a reboot when a kernel panic occurs. A *vmlinux* ELF kernel image is used as the capture kernel. Because *vmlinux* is a general ELF boot image and because kexec is theoretically OS agnostic, you need to specify via the `--args-linux` option that the following arguments have to be interpreted in a Linux-specific manner. The capture kernel boots asynchronously during a kernel crash, so device drivers using shared interrupts may fatally express their unhappiness during boot. To be nice to such drivers, specify `irqpoll` in the command line passed to the capture kernel using `--append`.

To use kexec with kdump, you need some additional kernel configuration settings. The capture kernel requires access to kernel memory of the crashed kernel to generate a full dump, so the latter cannot just overwrite the former as was done by kexec in the non-kdump case. The running kernel needs to reserve a memory region to run the capture kernel. To mark this region

- Boot the first kernel with the command-line argument `crashkernel=64M@16M` (or other suitable `size@start` values). Also include debug symbols in the kernel image by enabling `CONFIG_DEBUG_INFO` (*Kernel Hacking* ⟶ *Compile the Kernel with Debug Info*) in the configuration menu.

- While configuring the capture kernel, set `CONFIG_PHYSICAL_START` to the same `start` value assigned above (16M in this case). If you kexec into the capture kernel and peek inside */proc/meminfo*, you will find that `size` (64M in this case) is the total amount of physical memory that this kernel can see.

Now that you're comfortable with kexec and have mastered it from the perspective of a kdump user, let's delve into kdump and use it to analyze some real-world kernel crashes.

Kdump

An image of system memory captured after a kernel crash or hang is called a *crash dump.* Analyzing a crash dump can give valuable clues for postmortem analysis of kernel problems. However, obtaining a dump after a kernel crash is inherently unreliable because the storage driver responsible for logging data onto the dump device might be in an undefined state.

Until the advent of kdump, *Linux Kernel Crash Dump* (LKCD) was the popular mechanism to obtain and analyze dumps. LKCD uses a temporary dump device (such as the swap partition) to capture the dump. It then warm reboots back to a healthy state and copies the dump from the temporary device to a permanent location. A tool called *lcrash* is used to analyze the dump. The disadvantages with LKCD include the following:

- Even copying the dump to a temporary device might be unreliable on a crashed kernel.

- Dump device configuration is nontrivial.

- The reboot might be slow because swap space can be activated only after the dump has been safely saved away to a permanent location.

- LKCD is not part of the mainline kernel, so installing the proper patches for your kernel version is a hurdle.

Kdump is not burdened with these shortfalls. It eliminates indeterminism by collecting the dump after booting into a healthy kernel via kexec. Also, because memory state is preserved after a kexec reboot, the memory image can be accurately accessed from the capture kernel.

Let's first get the preliminary kdump setup out of the way:

1. Ask the running kernel to kexec into a capture kernel if it encounters a panic. The capture kernel should additionally have `CONFIG_HIMEM` and `CONFIG_CRASH_DUMP` turned on. (Both these options sit inside *Processor type and Features* in the kernel configuration menu.)

2. After the capture kernel boots, copy the collected dump information from `/proc/vmcore` (obtained by enabling `CONFIG_PROC_VMCORE` in the kernel configuration menu) to a file on your hard disk:

   ```
   bash> cp /proc/vmcore /dump/vmcore.dump
   ```

   You can also save other information such as the raw memory snapshot of the crashed kernel, via */dev/oldmem*.

3. Boot back into the first kernel. You are now ready to start dump analysis.

Let's use the collected dump file and the *crash* tool to analyze some example kernel crashes. Introduce this bug inside the interrupt handler of the RTC driver (*drivers/char/rtc.c*):

```
irqreturn_t rtc_interrupt(int irq, void *dev_id)
{
+  volatile int *integerp = 0xFF;
+  int integerd = *integerp;  /* Bad memory reference! */

   spin_lock(&rtc_lock);
   /* ... */
```

Trigger execution of the handler by enabling interrupts via the `hwclock` command:

```
bash> hwclock
...    ⟶ Kernel panic occurs when execution hits rtc_interrupt()
         causing kexec to boot into the capture kernel.
```

Save */proc/vmcore* to */dump/vmcore.dump*, reboot back into the first (crashed) kernel, and start analysis using the crash tool. In a real-world situation, of course, the dump might be captured at a customer site, whereas the

analysis is done at a support center:

```
bash> crash /usr/src/linux/vmlinux /dump/vmcore.dump
crash 4.0-2.24
...

      KERNEL: /usr/src/linux/vmlinux
    DUMPFILE: /root/vmcore.dumpfile
        CPUS: 1
        DATE: Mon Nov 26 04:15:49 2007
      UPTIME: 00:17:22
LOAD AVERAGE: 0.82, 1.02, 0.87
       TASKS: 63
       ...
       PANIC: "Oops: 0000 [#1]" (check log for details)
crash>
```

Examine the stack trace to understand the cause of the crash:

```
crash> bt
PID: 0      TASK: c03a32e0  CPU: 0   COMMAND: "swapper"
 #0 [c0431eb8] crash_kexec at c013a8e7
 #1 [c0431f04] die at c0103a73
 #2 [c0431f44] do_page_fault at c0343381
 #3 [c0431f84] error_code (via page_fault) at c010312d
    EAX: 00000008  EBX: c59a5360  ECX: c03fbf90  EDX: 00000000
    EBP: 00000000
    DS:  007b      ESI: 00000000  ES:  007b      EDI: c03fbf90
    CS:  0060      EIP: f8a8c004  ERR: ffffffff  EFLAGS: 00010092
 #4 [c0431fb8] rtc_interrupt at f8a8c004
 #5 [c0431fc4] handle_IRQ_event at c013de51
 #6 [c0431fdc] __do_IRQ at c013df0f
```

The stack trace points the needle of suspicion at `rtc_interrupt()`. Let's disassemble the instructions near `rtc_interrupt()`:

```
crash> dis 0xf8a8c000 5
0xf8a8c000 <rtc_interrupt>:     push   %ebx
0xf8a8c001 <rtc_interrupt+1>:   sub    $0x4,%esp
0xf8a8c004 <rtc_interrupt+4>:   mov    0xff,%eax
0xf8a8c009 <rtc_interrupt+9>:   mov    $0xc03a6640,%eax
0xf8a8c00e <rtc_interrupt+14>:  call   0xc0342300 <_spin_lock>
```

The instruction at address `0xf8a8c004` is attempting to move the contents of the `EAX` register to address `0xff`, which is clearly the invalid deference that caused the crash. Fix this and build a new kernel.

If you use the `irq` command, you can figure out the identity of the interrupt that was in progress during the time of the crash. In this case, the output confirms that the RTC interrupt was indeed active:

```
crash> irq
    IRQ: 8
 STATUS: 1 (IRQ_INPROGRESS)
...
```

```
...
handler: f8a8c000  <rtc_interrupt>
            flags: 20000000  (SA_INTERRUPT)
             mask: 0
             name: f8a8c29d  "rtc"

crash> quit
bash>
```

Let's now shift gears and look at a case where the kernel freezes, rather than generate an "oops." Consider the following buggy driver `init()` routine:

```
static int __init
mydrv_init(void)
{

  spin_lock(&mydrv_wq.lock);  /* Usage before initialization! */
  spin_lock_init(&mydrv_wq.lock);

  /* ... */
}
```

The code is erroneously using a spinlock before initializing it. Effectively, the CPU spins forever trying to acquire the lock, and the kernel appears to hang. Let's debug this problem using kdump. In this case, there will be no auto-trigger because there is no panic, so force a crash dump by pressing the magic Sysrq key combination, Alt-Sysrq-c. You may need to enable Sysrq by writing a 1 to */proc/sys/kernel/sysrq*:

```
bash> echo 1 > /proc/sys/kernel/sysrq
bash> insmod mydrv.ko
```

This induces the kernel to hang inside `mydrv_init()`. Press the Alt-Sysrq-c key combination to trigger a crash dump:

```
Alt-Sysrq-c
...                          → Messages announcing that a crash dump
                               has been triggered
```

Save the dump to disk after kexec boots the capture kernel, boot back to the original kernel, and run crash on the saved dump:

```
bash> crash vmlinux vmcore.dump
       ...
       PANIC: "SysRq : Trigger a crashdump"
         PID: 2115
     COMMAND: "insmod"
        TASK: f7c57000  [THREAD_INFO: f6170000]
         CPU: 0
       STATE: TASK_RUNNING (SYSRQ)
crash>
```

Test the waters by checking the identity of the process that was running at the time of the crash. In this case, it

was apparently *insmod*(of *mydrv.ko*):

```
crash> ps
    ...
    2171   2137  0  f6bb7000  IN  0.5   11728  5352  emacs-x
    2214      1  0  f6b5b000  IN  0.1    2732  1192  login
    2230   2214  0  f6bb0550  IN  0.1    4580  1528  bash
  > 2261   2230  0  c596f550  RU  0.0    1572   376  insmod
```

The stack trace doesn't yield much information other than telling you that a Sysrq key press was responsible for the dump:

```
crash> bt
PID: 2115   TASK: f7c57000  CPU: 0   COMMAND: "insmod"
 #0 [c0431e68] crash_kexec at c013a8e7
 #1 [c0431eb4] __handle_sysrq at c0254664
 #2 [c0431edc] handle_sysrq at c0254713
```

Let's next try peeking at the log messages generated by the crashed kernel. The `log` command reads the messages from the kernel `printk` ring buffer present on the dump file:

```
crash> log
...
BUG: soft lockup detected on CPU#0!

Pid: 2261, comm:               insmod
EIP: 0060:[<c010ec1b>] CPU: 0
EIP is at delay_pmtmr+0xb/0x20
EFLAGS: 00000246    Tainted: P       (2.6.16.16 #11)
EAX: 5caaa48c EBX: 00000001 ECX: 5caaa459 EDX: 00000012
ESI: 02e169c9 EDI: 00000000 EBP: 00000001 DS: 007b ES: 007b
CR0: 8005003b CR2: 08062017 CR3: 35e89000 CR4: 000006d0
 [<c01fede9>] __delay+0x9/0x10
 [<c0200089>] _raw_spin_lock+0xa9/0x150
 [<f893d00d>] mydrv_init+0xd/0xb2 [wqdrv]
 [<c0136565>] sys_init_module+0x175/0x17a2
 [<c015d834>] do_sync_read+0xc4/0x100
 [<c013ce4d>] audit_syscall_entry+0x13d/0x170
 [<c0105578>] do_syscall_trace+0x208/0x21a
 [<c0102f05>] syscall_call+0x7/0xb
SysRq : Trigger a crashdump
crash>
```

The log offers two useful pieces of debug information. First, it lets you know that a soft lockup was detected on the crashed kernel. As discussed in the section "Device Example: Watchdog Timer" in Chapter 5, "Character Drivers," the kernel detects soft lockups as follows: A kernel watchdog thread runs once a second and touches a per-CPU timestamp variable. If the CPU sits in a tight loop, the watchdog thread cannot update this timestamp. An update check is carried out during timer interrupts using `softlockup_tick()` (defined in *kernel/softlockup.c*). If the watchdog timestamp is more than 10 seconds old, it concludes that a soft lockup has occurred and emits a kernel message to that effect.

Second, the log frowns upon `mydrv_init()+0xd` (or `0xf893d00d`), so let's look at the disassembly of the surrounding code region:

```
crash> dis f893d000 5
dis: WARNING: f893d000: no associated kernel symbol found
0xf893d000:     mov    $0xf89f1208,%eax
0xf893d005:     sub    $0x8,%esp
0xf893d008:     call   0xc0342300 <_spin_lock>
0xf893d00d:     movl   $0xffffffff,0xf89f1214
0xf893d017:     movl   $0xffffffff,0xf89f1210
```

The return address in the stack is `0xf893d00d`, so the kernel is hanging inside the previous instruction, which is a call to `spin_lock()`. If you co-relate this with the earlier source snippet and look at it in the eye, you can see the error sequence, `spin_lock()/spin_lock_init()`, staring sorrowfully back at you. Fix the problem by swapping the sequence.

You may also use crash to peek at data structures of interest, but be aware that memory regions that were swapped out during the crash are not part of the dump. In the preceding example, you can examine `mydrv_wq` as follows:

```
crash> rd mydrv_wq 100
f892c200:  00000000 00000000 00000000 00000000   ................
...
f892c230:  2e636373 00000068 00000000 00000011   scc.h...........
```

Gdb is integrated with crash, so you can pass commands from crash to gdb for evaluation. For example, you can use gdb's `p` command to print data structures.

Looking at the Sources

Architecture-dependent portions of kexec reside in *arch/your-arch/kernel/machine_kexec.c* and *arch/your-arch/kernel/relocate_kernel.S*. The generic parts live in *kernel/kexec.c* (and *include/linux/kexec.h*). Peek inside *arch/your-arch/kernel/crash.c* and *arch/your-arch/kernel/crash_dump.c* for the kdump implementation. *Documentation/kdump/kdump.txt* contains installation information.

Profiling

Profiling points you to those regions of code that burn more CPU cycles. Profilers help sense the presence of code bottlenecks and come in different flavors. The *OProfile* kernel profiler, included with the 2.6 kernel, uses hardware assist to gather profile data. The *gprof* application profiler, on the other hand, relies on compiler assist to collect profiling information.

Kernel Profiling with OProfile

OProfile samples data at regular intervals using hardware performance counters supported by many processors. The performance counters can be programmed to count events such as the number of cache misses. On systems where the processor does not support performance counters, OProfile obtains limited information by collecting data during timer events.

OProfile consists of the following:

- A kernel layer that collects profiling information.[7] To enable OProfile in your kernel, enable `CONFIG_PROFILING`, `CONFIG_OPROFILE`, and `CONFIG_APIC` and recompile.

  [7] If you are still using a 2.4 kernel, you have to patch your kernel sources with OProfile support.

- The *oprofiled* daemon.

- A suite of post-profiling tools such as *opcontrol*, *opreport*, and *op_help* that help in detailed analysis of the collected data. These tools are included with several distributions; if your distribution doesn't have them, however, you can download precompiled binaries.

To illustrate the basics of kernel profiling, let's simulate a bottleneck in the filesystem layer and use OProfile to detect it. Our code area of interest is the portion of the filesystem layer that reads directories (function `vfs_readdir()` in *fs/readdir.c*)

First, use opcontrol to configure OProfile:

```
bash> opcontrol --setup --vmlinux=/path/to/kernelsources/vmlinux
                --event=GLOBAL_POWER_EVENTS:100000:1:1:1
```

The event specifier asks OProfile to collect samples during `GLOBAL_POWER_EVENTS` (time during which the processor is not stopped). The numerals adjacent to the event specifier denote the sampling count in clock cycles, unit mask filter, kernel-space counting, and user-space counting, respectively. If you would like to sample `x` times every second and your processor is running at a frequency of `cpu_speed` HZ, your sample count should approximately be `(cpu_speed/x)`. A larger count generates a finer profile but also results in more CPU overhead.

The events supported by OProfile depend on your processor:

```
bash> opcontrol -l  ⟶ List available events on a Pentium 4 CPU
GLOBAL_POWER_EVENTS: (counter: 0, 4)
```

```
        time during which processor is not stopped (min count: 3000)
BRANCH_RETIRED: (counter: 3, 7)
        retired branches (min count: 3000)
MISPRED_BRANCH_RETIRED: (counter: 3, 7)
        retired mispredicted branches (min count: 3000)
BSQ_CACHE_REFERENCE: (counter: 0, 4)
...
```

Next, start OProfile and run a benchmarking tool that stresses those parts of the kernel you would like to profile. Look at http://lbs.sourceforge.net/ for a list of benchmarking projects on Linux. For this example, let's exercise the *Virtual File System* (VFS) layer by recursively listing all files in the system:

```
bash> opcontrol --start    → Start data collection
bash> ls -lR /             → Stress test
bash> opcontrol --dump     → Save profiled data
```

Use opreport to look at the profiling results. The % column provides a measure of the function's load on the system:

Code View:

```
bash> opreport -l /path/to/kernelsources/vmlinux

CPU: P4 / Xeon, speed 2992.9 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor
is not stopped) with a unit mask of 0x01 (count cycles when processor is active)
count 100000
samples  %        symbol name
914506   24.2423  vgacon_scroll    → ls output printed to console
406619   10.7789  do_con_write
273023    7.2375  vgacon_cursor
206611    5.4770  __d_lookup
...
1380      0.0366  vfs_readdir      → Our routine of interest
...
1        2.7e-05  vma_prio_tree_next
```

Let's now simulate a bottleneck in the VFS code by introducing a 1-millisecond delay in `vfs_readdir()`. This is done in Listing 21.6.

Listing 21.6. vfs_readdir() Defined in fs/read_dir.c

```
int vfs_readdir(struct file *file, filldir_t filler, void *buf)
{
  struct inode *inode = file->f_ dentry->d_inode;
  int res = -ENOTDIR;

+ /* Introduce a millisecond bottleneck
+    (HZ is set to 1000 on this system) */
+ unsigned long timeout = jiffies+1;
+ while (time_before(jiffies, timeout));
+ /* End of bottleneck */

  /* ... */
}
```

Compile the kernel with this change and recollect the profile. The new data looks like this:

Code View:

```
bash> opreport -l /path/to/kernelsources/vmlinux

CPU: P4 / Xeon, speed 2993.08 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped)
with a unit mask of 0x01 (count cycles when processor is active)
count 100000
samples  %        symbol name
6178015  57.1640  vfs_readdir     → Our routine of interest
1065197  9.8561   vgacon_scroll   → ls output printed to console
479801   4.4395   do_con_write
...
```

As you can see, the bottleneck is clearly reflected in the profiled data. `vfs_readdir()` has now jumped to the top of the list!

You can use OProfile to obtain a lot more information. You can, for example, gather the percentage of data cache line misses. Caches are fast memory close to the processor. Fetches to cache are done in units of the processor cache line (32 bytes for Pentium 4). If the data you need to access is not already present in the cache (a cache miss), the processor has to fetch it from main memory, and this burns more CPU cycles. Subsequent accesses to that memory (and the surrounding bytes touched into the cache) will be faster until the corresponding cache line gets invalidated. You can configure OProfile to count the number of cache misses by profiling your kernel code for the `BSQ_CACHE_REFERENCE` event (for Pentium 4). You can then tune your code, possibly by realigning fields in data structures, to achieve better cache utilization:

Code View:

```
bash> opcontrol --setup
                --event=BSQ_CACHE_REFERENCE:50000:0x100:1:1
                --vmlinux=/path/to/kernelsources/vmlinux
                                  → Unit mask 0x100 denotes an L2 cache miss
bash> opcontrol --start           → Start data collection
bash> ls -lR /                    → Stress
bash> opcontrol --dump            → Save profile
bash> opreport -l /path/to/kernelsources/vmlinux
CPU: P4 / Xeon, speed 2993.68 MHz (estimated)
```

```
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit) with a
unit mask of 0x100 (read 2nd level cache miss) count 50000
samples  %        symbol name
73       29.6748  find_inode_fast
59       23.9837  __d_lookup
27       10.9756  inode_init_once
...
```

If you run OProfile on different kernel versions and look at the corresponding change logs, you might be able to figure out reasons for code changes in different parts of the kernel.

You have only touched the surface of what can be accomplished using OProfile. For more information, visit http://oprofile.sourceforge.net/.

Application Profiling with Gprof

If you need to profile only an application process in isolation without profiling the kernel code that might get executed on its behalf, use *gprof* rather than OProfile. Gprof relies on additional code generated by the compiler to profile C, Pascal, or Fortran programs. Let's use gprof to profile the following code snippet:

```
main(int argc, char *argv[])
{
  int i;

  for (i=0; i<10; i++) {
    if (!do_task()) {      /* Perform task */
      do_error_handling(); /* Handle errors */
    }
  }
}
```

Use the `-pg` option to ask the compiler to include extra code that generates a call graph profile when the program runs. The `-g` option generates symbolic information:

```
bash> gcc -pg -g -o myprog myprog.c
bash> ./myprog
```

This produces *gmon.out*, which is a call graph of *myprog*. Run gprof to view the profile:

```
bash> gprof -p -b myprog

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call   name
 65.17     2.75     2.75        2     1.38     1.38   do_error_handling
 34.83     4.22     1.47       10     0.15     0.15   do_task
```

This shows that the error path was hit twice during execution. You can tune the code to produce fewer

traversals of the error path and rerun gprof to generate an updated profile.

Tracing

Tracing provides insight into behavioral problems that manifest during interactions between different code modules. A common way to obtain execution traces is by using `printk`s. While `printk` is perhaps the most heavily used method for kernel debugging (there are more than 62,000 `printk()` statements in the 2.6.23 source tree), it is not sophisticated enough for high-volume tracing. *Linux Trace Toolkit* (LTT) is a powerful tool that lets you obtain complex system level traces with minimum overhead.

Linux Trace Toolkit

LTT extracts execution traces that are useful for postmortem analysis and is valuable in situations where it may not be possible to use a debugger. Unlike OProfile, which collects data by sampling events at regular intervals, LTT provides exact traces of events as and when they occur.

LTT consists of four components:

- A core module that provides trace services to the rest of the kernel. The collected traces are copied to a kernel buffer.
- Code that makes use of the trace services. These are inserted at points where you want to capture trace dumps.
- A trace daemon that pulls trace information from the kernel buffer to a permanent location in the filesystem.
- Utilities such as *tracereader* and *tracevisualizer* that interpret raw trace data and convert it into human-readable form. If you are developing code for an embedded device having no GUI support, you can transparently run these tools on another machine.

LTT is not part of the mainline kernel.[8] You may download LTT kernel patches, trace daemon, and user-space trace utilities from www.opersys.com/LTT.

[8] LTT was included as a release candidate in the *2.6.11-rc1-mm1* patch, downloadable from www.kernel.org.

Let's find out what LTT offers with the help of a simple example. Assume that you are seeing data corruption when your application is reading from a device. You first want to figure out whether the device is sending bad data or whether a kernel layer is introducing the corruption. To do that, dump data packets and data structures at the device driver level. Listing 21.7 initializes the LTT events that you plan to generate.

Listing 21.7. Creating LTT Events

```
#include <linux/trace.h>

int data_packet, driver_data; /* Trace events */

/* Driver init */
static int __init mydriver_init(void)
{
  /* ... */

  /* Event to dump packets received from the device */
  data_packet = trace_create_event("data_pkt",
                                   NULL,
                                   CUSTOM_EVENT_FORMAT_TYPE_HEX,
                                   NULL);

  /* Event to dump a driver structure */
  driver_data = trace_create_event("dvr_data",
                                   NULL,
                                   CUSTOM_EVENT_FORMAT_TYPE_HEX,
                                   NULL);

  /* ... */

}
```

Next, let's add trace hooks to dump received packets and data structures when the driver reads data from the device. This is done in Listing 21.8 in the driver `read()` method.

Listing 21.8. Obtaining Trace Dumps

Code View:

```
struct mydriver_data driver_data; /* Private device structure */

/* Driver read() method */
ssize_t
mydriver_read(struct file *file, char *buf,
              size_t count, loff_t *ppos)
{
  char *buffer;

  /* Read numbytes bytes of data from the device into
     buffer */
  /* ... */

  /* Dump data Packet. If you see the problem only
     under certain conditions, say, when the packet length is
     greater than a value, use that as a filter */
  if (condition) {
    /* See Listing 21.7 for the definition of data_packet*/
    trace_raw_event(data_packet, numbytes, buffer);
  }

  /* Dump driver data structures */
  if (some_other_condition) {
    /* See Listing 21.7 for the definition of driver_data */
    trace_raw_event(driver_data, sizeof(driver_data), &driver_data);
  }
```

```
  /* ... */
}
```

Compile and run this code as part of the kernel or as a module. Remember to turn on trace support in the kernel by setting `CONFIG_TRACE` while configuring the kernel. The next step is to start the trace daemon:

```
bash> tracedaemon -ts60 /dev/tracer mylog.txt mylog.proc
```

/dev/tracer is the interface used by the trace daemon to access the trace buffer, `-ts60` asks the daemon to run for 60 seconds, *mylog.txt* is the file where you want to store the generated raw trace, and *mylog.proc* is where you want to save the system state obtained from procfs. Later versions of LTT use a mechanism called *relayfs* rather than the */dev/tracer* device for efficiently transferring data from the kernel trace buffer to user space.

Run your application that reads data from the device:

```
bash> ./application  ⟶ Trigger invocation of mydriver_read()
```

mylog.txt should now contain the requested trace data. The generated raw trace can be analyzed using the *tracevisualizer* tool. Choose the *Custom Events* option and search for `data_pkt` and `dvr_data` events. The output looks like this:

```
#####################################################################
Event      Time SECS    MICROSEC   PID     Length  Description
#####################################################################
data_pkt   1,110,563,008,742,457    0       27      12 43 AB AC 00 01 0D 56
data_pkt   1,110,563,008,743,151    0       27      01 D4 73 F1 0A CB DD 06
dvr_data   1,110,563,008,743,684    0       25      0D EF 97 1A 3D 4C
...
```

The last column holds the trace data. The timestamp shows the instant when the trace was collected. If the data looks corrupt, the device could be sending bad data. Otherwise, the problem must be in a higher kernel layer and can be further isolated by obtaining traces from a different point in the data-flow path.

The next generation of LTT called LTTng is available for download from http://ltt.polymtl.ca/. This project also includes a post-trace analyzer called *Linux Trace Toolkit Viewer* (LTTV).

If your need is only to perform limited tracing of a user application, you can use the *strace* utility rather than LTT. Strace uses the *ptrace* support in the kernel to intercept system calls. It prints out a list of system calls made by your application, along with the corresponding arguments and return values.

Linux Test Project

Linux Test Project (LTP), hosted at http://ltp.sourceforge.net/, is a suite consisting of around 3,000 tests designed to exercise different parts of the kernel. Most tests run without user intervention. Others such as networking and storage media tests need some manual configuration.

Download the source tar ball from the LTP home page, run *make*, and invoke the wrapper script *runltp* from the root of the source tree to start the tests. To capture the results in *logfile* in the *results/* directory, do this:

```
bash> runltp -p -l logfile
```

Some errors generated by LTP are "expected." The LTP website documents the list of expected errors for various kernel versions. Also in the website is an interesting analysis of LTP's code coverage (overall coverage, lines in path, and distinct lines hit) for a few kernel versions, split across directories in the kernel tree.

User Mode Linux

User Mode Linux (UML), hosted at http://user-mode-linux.sourceforge.net/, lets you debug the kernel without "oops"ing the machine. To accomplish this, an instance of the kernel (called the *guest* kernel) runs as a user mode process over the running kernel (called the *host* kernel).

UML has diverse users. It can function as an environment for testing kernel and application code, a vehicle to experiment with unstable kernels, a secure pseudo computer for hosting services such as web servers, or a tool to learn Linux internals. UML is more useful for debugging hardware-independent portions of the kernel than for device driver debugging.

Diagnostic Tools

The *sysfsutils* package helps you navigate the voluminous amount of data present inside sysfs. This, and other Linux diagnostic tools such as *sysdiag* and *lsvpd*, can be downloaded from http://linux-diag.sourceforge.net/.

Kernel Hacking Config Options

Several options exist under *Kernel hacking* in the kernel configuration menu that can emit valuable debug information. If you enable an option, corresponding debug code gets compiled when you build the kernel.[9] Here are a few examples:

[9] Some kernel hacking options are architecture-dependent.

1. *Show Timing information on printks* (`CONFIG_PRINTK_TIME`) adds timing instrumentation to `printk()` output, so you can use `printk`s as checkpoints for measuring execution times and identifying slow code regions.

2. Using freed memory results in memory poisoning. *Debug slab memory allocations* (`CONFIG_DEBUG_SLAB`) helps you detect such problems.

3. *Spinlock and rw-lock debugging: basic checks* (`CONFIG_DEBUG_SPINLOCK`) finds lock-related problems such as uninitialized spinlock usage and helps catch code that is not SMP-safe.

4. You have already worked with *Magic SysRq key*(`CONFIG_MAGIC_SYSRQ`) when you learned to use kdump. If you turn this on, you will have some avenues left even if the kernel crashes during debugging. For example, pressing Alt-Sysrq-t produces a dump of current tasks, whereas Alt-Sysrq-p prints the contents of processor registers.

5. *Detect Soft Lockups* (`CONFIG_DETECT_SOFTLOCKUP`) utilizes the services of a watchdog to detect tight loops in kernel code that last for more than 10 seconds. We looked at this when we analyzed a kernel hang using kdump. Note that hardware lockups cannot be found this way. For that, use the services of a Non-Maskable Interrupt (NMI)-watchdog if your platform supports it.

6. If you enable `CONFIG_DEBUG_SLAB`, `CONFIG_DEBUG_HIMEM`, or `CONFIG_DEBUG_PAGE_ALLOC` while configuring your kernel, additional error-checking code gets compiled that help debug problems related to memory management.

7. *Check for stack overflows* (`CONFIG_DEBUG_STACKOVERFLOW`) adds code to emit warnings if the available stack space falls below a threshold. *Stack utilization instrumentation* (`CONFIG_DEBUG_STACK_USAGE`) adds stack space instrumentation to the magic Sysrq key output. Another related option, `CONFIG_4KSTACKS`, lets you set the kernel stack size to 4KB rather than 8KB.

8. *Verbose BUG() reporting* (`CONFIG_DEBUG_BUGVERBOSE`) produces extra debug information when any kernel code invokes `BUG()`, assuming that you have `CONFIG_BUG` turned on during kernel configuration.

Some debug options live outside the *Kernel hacking* submenu, too. For example, we enabled `CONFIG_KALLSYMS` in this chapter to debug an "oops" message using gdb and to kprobe a kernel module. This option lives under *General setup* ⟶ *Configure Standard Kernel Features (for small systems)* in the configuration menu.

Kernel hacking options result in overhead and increased footprint, so don't leave them on in production-level kernels.

Test Equipment

It goes without saying that you need the full complement of relevant test equipment for device driver debugging. If you are testing a modem interface in a digital-only laboratory environment for example, you will be well served by a phone simulator. If a high-speed serial driver is manifesting parity errors, an oscilloscope will aid your problem analysis. If you are writing an I/O device driver, it will help if you have the associated bus analyzer. If you are writing a network driver, the corresponding protocol line sniffer will ease your debugging effort.

Chapter 22. Maintenance and Delivery

In This Chapter

- Coding Style 642
- Change Markers 642
- Version Control 643
- Consistent Checksums 643
- Build Scripts 645
- Portable Code 647

You have reached the end of the device driver tour, but implementing a driver is only a part of the software development life cycle. Before wrapping up, let's discuss a few ideas that contribute to operational efficiency during software maintenance and delivery.

Coding Style

The life span of many Linux devices range from 5 to 10 years, so adherence to a standard coding style helps support the product long after you have moved out of the project.

A powerful editor coupled with an organized writing style makes it easier to correlate code with thought. There can be no infallible guidelines for good style because it's a matter of personal preference, but a uniform manner of coding is invaluable if there are multiple developers working on a project.

Agree on common coding standards with team members and the customer before starting a project. The coding style preferred by kernel developers is described in *Documentation/CodingStyle* in the source tree.

Chapter 22. Maintenance and Delivery

In This Chapter

- Coding Style 642
- Change Markers 642
- Version Control 643
- Consistent Checksums 643
- Build Scripts 645
- Portable Code 647

You have reached the end of the device driver tour, but implementing a driver is only a part of the software development life cycle. Before wrapping up, let's discuss a few ideas that contribute to operational efficiency during software maintenance and delivery.

Coding Style

The life span of many Linux devices range from 5 to 10 years, so adherence to a standard coding style helps support the product long after you have moved out of the project.

A powerful editor coupled with an organized writing style makes it easier to correlate code with thought. There can be no infallible guidelines for good style because it's a matter of personal preference, but a uniform manner of coding is invaluable if there are multiple developers working on a project.

Agree on common coding standards with team members and the customer before starting a project. The coding style preferred by kernel developers is described in *Documentation/CodingStyle* in the source tree.

Change Markers

Using a marker such as `CONFIG_MYPROJECT` to tag additions and deletions to existing kernel source files helps highlight project-specific changes to the source tree. One can recursively grep for the marker string from the root of the code tree to learn the location of all kernel changes implemented for the project. The following example marks code changes to *drivers/i2c/busses/i2c-i801.c*. The modification introduces a check for a new PCI device ID during setup and eliminates a configuration byte access:

```
/* ... */
switch(dev->device) {
  case PCI_DEVICE_ID_INTEL_82801DB_3:
#if defined (CONFIG_MYPROJECT)
   case PCI_DEVICE_ID_MYID :
#endif
  /* ... */
}
/* ... */
#if !defined (CONFIG_MYPROJECT)
   pci_write_config_byte(I801_dev, SMBHSTCFG, temp);
#endif
return 0;
/* ... */
```

`CONFIG_MYPROJECT` also functions as a configuration-time switch to enable or disable project-specific changes.

It's a good idea to have submarkers for various subtasks in your project. So, if you are modifying the kernel for fast boot as part of your project, wrap those changes within a submarker such as `CONFIG_MYPROJECT_FASTBOOT`.

Version Control

You can't execute a serious project without the services of a robust version control repository. A version control system helps manage multiple versions of source code and regulates file accesses by team members. Concurrent Versions System or CVS (www.nongnu.org/cvs) is an open source revision control software that has been around for a long time and comes bundled with many Linux distributions. Another versioning system called subversion (http://subversion.tigris.org) was developed as an intended replacement for CVS. Git (http://git.or.cz) is the version control system of choice for kernel developers and is used to maintain several open source projects, including the Linux kernel. Ample documentation on these systems is available on the Internet.

Consistent Checksums

Because of legal issues latent in distributing the kernel, companies often ship kernel modifications to customers in the form of a source patch generated against an agreed-upon base. Customers, in turn, integrate the patch into an in-house code repository and build the software locally.

Comparing the MD5 checksum of your binary images with that of your customer's is a guard against patching errors, but the values may not match as-is because the kernel and module images often contain information specific to the build environment. To force identical MD5 sums, exclude such data while generating kernel and module images at either end. Here are some typical scenarios that inject environmental data into the object image:

- Some driver methods toss a call to `BUG()` to announce conditions that are never supposed to occur. `BUG()` spits out, among other things, the offending filename and line number. The pathname of the file depends on the location of your build sandbox. It gets imprinted in the produced image and contributes to MD5 variance. For example, look at `nfs_unlock_request()` in *fs/nfs/pagelist.c*:

  ```
  void
  nfs_unlock_request(struct nfs_page *req)
  {
    if (!NFS_WBACK_BUSY(req)) {
      printk(KERN_ERR "NFS: Invalid unlock attempted\n");
      BUG();
    }
    /* ... */
  }
  ```

  `BUG()` is defined in *include/asm-your-arch/bug.h:*

  ```
  #define BUG() do {\
  __asm__ __volatile__ ("ud2\n"\
                        ...
                        : : "I" (__LINE__), "I"(__FILE__))
  ```

  You can compile `BUG()` away by disabling `CONFIG_BUG` during kernel configuration. Or you may get rid of the line number and filename information emitted by `BUG()` by switching off `CONFIG_DEBUG_BUGVERBOSE`.

- The wd33c93 driver (*drivers/scsi/wd33c93.c*) announces the time of compilation during initialization. You will find this snippet if you go to the end of the initialization routine, `wd33c93_init()`:

  ```
  void
  wd33c93_init(struct Scsi_Host *instance,
               const wd33c93_regs regs, dma_setup_t setup,
               dma_stop_t stop, int clock_freq)
  {
    /* ... */
    printk("    Version %s - %s, Compiled %s at %s\n",
    WD33C93_VERSION, WD33C93_DATE, __DATE__, __TIME__);
  }
  ```

  The build timestamp thus gets embedded inside the image, causing the MD5 checksum to depend on it.

- The `CONFIG_IKCONFIG_PROC` configuration option, if enabled, introduces the configuration timestamp in the kernel image. This information is available as part of */proc/config.gz* at runtime.

- Utilities living inside the *scripts/* directory in the kernel tree also contribute to MD5 variance by injecting the output of programs such as hostname, date, whoami and domainname, into kernel header files such as *include/linux/ -compile.h*.

Hunt down and mask out such direct and indirect references to environmental information to generate identical checksums at both ends. Of course, you need not bother about kernel modules that aren't relevant. Envelope your code modifications within a change marker such as `CONFIG_MYPROJECT_SAME_MD5` and create a kernel configuration switch to turn consistent MD5 generation on or off. When you finish, run md5sum on the stripped vmlinux image.

Build Scripts

Customers generally ask for periodic software builds during the development cycle. Each build includes new features or bug fixes. The deliverables for an embedded PC derivative, for example, may include firmware components such as the base kernel, loadable device driver modules, filesystem utilities, bootloader, BIOS, and on-card microcode. To automate build generation, it's a good idea to implement a set of versatile build scripts that obtain a source code snapshot from the version control repository and generate a packaged deliverable.

Listing 22.1 shows a skeletal build script that assumes use of CVS for version control. This is a simple script that shows only the kernel build. In the real world, you might need a sophisticated suite of scripts that package several software components and manage different installation scenarios.

Listing 22.1. A Simple Build Script

Code View:

```
# Check that compilation tools are installed
#...

# Assume that $user contains the user name, $cvsserver contains
# the CVS server name and /path/to/repository is the location
# of your project's repository on the CVS server
CVS="cvs –d :pserver:$user@$cvsserver:/path/to/repository"
$CVS login

# Check-out the kernel
$CVS checkout kernel

# Build the kernel
cd kernel
make mrproper
#Get the .config file for your platform
cp arch/your-arch/configs/your_platform_defconfig .config
make oldconfig
make –j5 bzImage # Accelerate by spawning 5 instances of 'make'
if [ $? != 0 ]
then
  echo "Error building Kernel. Bailing out.."
  exit 1
fi

# Copy the kernel image to a target directory
cp arch/x86/boot/bzImage /path/to/target_directory/productname.kernel

# Build modules and install them in an appropriate directory
make modules
if [ $? != 0 ]
then
  echo "Error building modules. Bailing.."
  exit 2
fi

export INSTALL_MOD_PATH=»$TARGET_DIRECTORY/modules»
make modules_install

# Rebuild after forcing generation of identical MD5 sums and
```

```
# package the resulting checksum information.
#...

# Generate a source patch from the base starting point, assuming
# that KERNELBASE is the CVS tag for the vanilla kernel
cvs rdiff –u –r KERNELBASE kernel > patch.kernel

# Generate a changelog using "cvs log"
#...

# Package everything nicely into a tar ball
#...
```

After you satisfactorily complete build verification tests on the generated deliverable, initiate a post-build process to tag the current state of the version control system with a build identifier. This essentially attaches a name to the source snapshot corresponding to the build and helps trace problems to code versions. You can check out source versions based on the relevant build identifier when you later attempt to re-create reported field problems in your lab.

Portable Code

Portability directly translates to code reusability and easier maintenance. This is significant in today's marketplace, where there are an assortment of processors and innumerable peripheral chipsets. Things will fast spin out of control if you have to code separate bus drivers for each processor and different client device drivers for each host controller. Here are some hints for writing portable drivers:

- Make portability a design goal while architecting your driver.
- Using appropriate kernel APIs automatically injects a degree of portability. A USB driver using the services of the USB core is rendered independent of the USB host controller. It will work unchanged on different systems, irrespective on whether they use UHCI, OHCI, or something else.
- Write SMP-safe code.
- Write code that is 64-bit clean. Do not, for example, assign a pointer to an integer, even with valid typecasts.
- Write drivers such that they can be easily adapted for other similar devices.
- Use architecture-independent APIs wherever available. For example, calls to `outb()` or `inb()` will work irrespective of whether the processor uses I/O-mapped or memory-mapped addressing. If you do need to use architecture-specific code such as inline assembly, stow it away inside the appropriate *arch/your-arch/* directory.
- Push policy to header files and user space. Use macros and definitions wherever suitable.

Chapter 23. Shutting Down

In This Chapter

- Checklist 650
- What Next? 651

Before transitioning to init runlevel 0, let's summarize how to set forth on your way to Linux-enablement when you get hold of a new device. Here's a quick checklist.

Checklist

1. Identify the device's functionality and interface technology. Depending on what you find, review the chapter describing the associated device driver subsystem. As you learned, almost every driver subsystem on Linux contains a core layer that offers driver services, and an abstraction layer that renders applications independent of the underlying hardware (revisit Figure 18.3 in Chapter 18, "Embedding Linux"). Your driver needs to fit into this framework and interact with other components in the subsystem. If your device is a modem, learn how the UART, tty, and line discipline layers operate. If your chip is an RTC or a watchdog, learn how to conform to the respective kernel APIs. If what you have is a mouse, find out how to tie it with the input event layer. If your hardware is a video controller, glean expertise on the frame buffer subsystem. Before embarking on driving an audio codec, investigate the ALSA framework.

2. Obtain the device's data sheet and understand its register programming model. For an I^2C DVI transmitter, for example, get the device's slave address and the programming sequence for initialization. For an SPI touch controller, understand how to implement its finite state machine. For a PCI Ethernet card, find out the configuration space semantics. For a USB device, figure out the supported endpoints and learn how to communicate with them.

3. Search for a starting point driver inside the mighty kernel source tree. Research candidate drivers and hone in on a suitable one. Certain subsystems offer skeletal drivers that you can model after, if you don't find a close match. Examples are *sound/drivers/dummy.c, drivers/usb/usb-skeleton.c, drivers/net/pci-skeleton.c*, and *drivers/video/skeletonfb.c*.

4. If you obtain a starting point driver, investigate the exact differences between the associated device and your hardware by comparing the respective data sheets and schematics. For illustration, assume that you are putting Linux on a custom board that is based on a distribution-supported reference hardware. Your distribution includes the USB controller driver that is tested on the reference hardware, but does your

custom board use different USB transceivers? You have a frame buffer driver for the LCD controller, but does your board use a different display panel interface such as LVDS? Perhaps an EEPROM that sat on the I^2C bus on the reference board now sits on a 1-wire bus. Is the Ethernet controller now connected to a different PHY chip or even to a Layer 2 switch chip? Or perhaps the RS-232 interface to the UART has given way to RS-485 for better range and fidelity.

5. If you don't have a close starting point or if you decide to write your own driver from scratch, invest time in designing and architecting the driver and its data structures.

6. Now that you have all the information you need, arm yourself with software tools (such as ctags, cscope, and debuggers) and lab equipment (such as oscilloscopes, multimeters, and analyzers) and start writing code.

Chapter 23. Shutting Down

In This Chapter

- Checklist 650
- What Next? 651

Before transitioning to init runlevel 0, let's summarize how to set forth on your way to Linux-enablement when you get hold of a new device. Here's a quick checklist.

Checklist

1. Identify the device's functionality and interface technology. Depending on what you find, review the chapter describing the associated device driver subsystem. As you learned, almost every driver subsystem on Linux contains a core layer that offers driver services, and an abstraction layer that renders applications independent of the underlying hardware (revisit Figure 18.3 in Chapter 18, "Embedding Linux"). Your driver needs to fit into this framework and interact with other components in the subsystem. If your device is a modem, learn how the UART, tty, and line discipline layers operate. If your chip is an RTC or a watchdog, learn how to conform to the respective kernel APIs. If what you have is a mouse, find out how to tie it with the input event layer. If your hardware is a video controller, glean expertise on the frame buffer subsystem. Before embarking on driving an audio codec, investigate the ALSA framework.

2. Obtain the device's data sheet and understand its register programming model. For an I^2C DVI transmitter, for example, get the device's slave address and the programming sequence for initialization. For an SPI touch controller, understand how to implement its finite state machine. For a PCI Ethernet card, find out the configuration space semantics. For a USB device, figure out the supported endpoints and learn how to communicate with them.

3. Search for a starting point driver inside the mighty kernel source tree. Research candidate drivers and hone in on a suitable one. Certain subsystems offer skeletal drivers that you can model after, if you don't find a close match. Examples are *sound/drivers/dummy.c, drivers/usb/usb-skeleton.c, drivers/net/pci-skeleton.c*, and *drivers/video/skeletonfb.c*.

4. If you obtain a starting point driver, investigate the exact differences between the associated device and your hardware by comparing the respective data sheets and schematics. For illustration, assume that you are putting Linux on a custom board that is based on a distribution-supported reference hardware. Your distribution includes the USB controller driver that is tested on the reference hardware, but does your

custom board use different USB transceivers? You have a frame buffer driver for the LCD controller, but does your board use a different display panel interface such as LVDS? Perhaps an EEPROM that sat on the I^2C bus on the reference board now sits on a 1-wire bus. Is the Ethernet controller now connected to a different PHY chip or even to a Layer 2 switch chip? Or perhaps the RS-232 interface to the UART has given way to RS-485 for better range and fidelity.

5. If you don't have a close starting point or if you decide to write your own driver from scratch, invest time in designing and architecting the driver and its data structures.

6. Now that you have all the information you need, arm yourself with software tools (such as ctags, cscope, and debuggers) and lab equipment (such as oscilloscopes, multimeters, and analyzers) and start writing code.

What Next?

Linux is here to stay, but internal kernel interfaces tend to get fossilized as soon as someone figures out a cleverer way of doing things. No kernel code is etched in stone. As you learned, even the scheduler, considered sacred, has undergone two rewrites since the 2.4 days. The number of new lines of code appearing in the kernel tree runs into the millions each year. As the kernel evolves, new features and abstractions keep getting added, programming interfaces redesigned, subsystems restructured for extracting better performance, and reusable regions filtered into common cores.

You now have a solid foundation, so you can adapt to these changes. To maintain your cutting-edge, refresh your kernel tree regularly, browse the kernel mailing list frequently, and write code whenever you can. Linux is the future, and being a kernel guru pays. Stay at the front lines!

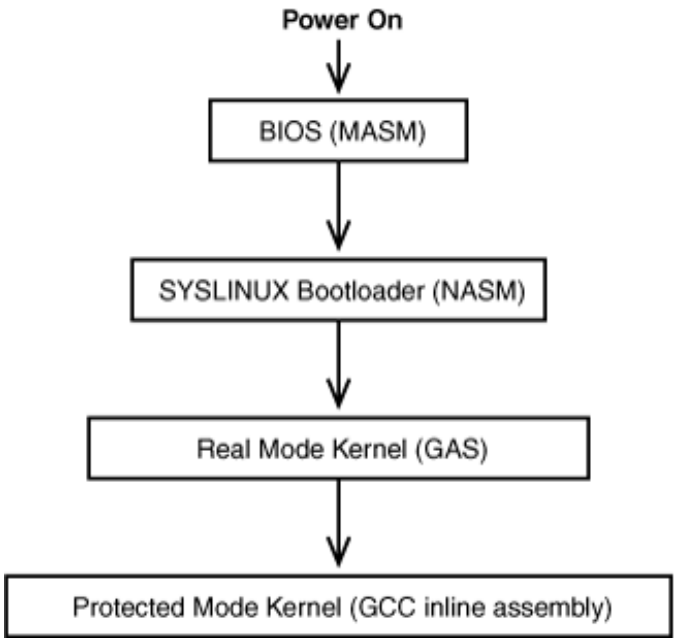
Appendix A. Linux Assembly

Device drivers sometimes need to implement some code snippets in assembly, so let's take a look at the different facets of assembly programming on Linux.

Figure A.1 shows the Linux boot sequence on a PC-compatible system and is a simpler version of Figure 2.1 in Chapter 2, "A Peek Inside the Kernel." The firmware components in the figure are implemented using different assembly syntaxes:

- The BIOS is typically written wholly in assembly. Some of the popular PC BIOSes are coded using assemblers such as the *Microsoft Macro Assembler* (MASM).
- Linux bootloaders such as LILO and GRUB are implemented using a mix of C and assembly. The SYSLINUX bootloader is entirely written in assembly using the *Netwide Assembler* (NASM).
- Real mode Linux startup code uses the *GNU Assembler* (GAS).
- Protected mode BIOS invocations are done in *inline assembly*, which is a construct supported by GCC to insert assembly in between C statements.

Figure A.1. Firmware components and assembly syntaxes.



In Figure A.1, the top two components generally follow Intel-based assembly syntax, whereas the bottom two are coded in AT&T (or GAS) syntax. There are exceptions; the assembly parts of GRUB use GAS.

To illustrate the differences between these two syntaxes, consider code that outputs a byte to the parallel port. In Intel format used by the BIOS or the bootloader, you would write the following:

```
mov dx, 03BCh   ;0x3BC is the I/O address of the parallel port
mov al, 0ABh    ;0xAB is the data to be output
out dx, al      ;Send data to the parallel port
```

However, if you want to perform the same task from Linux real mode startup code, you need to do this:

```
movw $0x3BC, %dx
movb $0xAB,  %al
outb %al,    %dx
```

You can see that unlike in Intel format, in AT&T syntax, the source operand comes first, and the destination operand comes second. Register names in AT&T format are preceded by `%`, and immediate operands are preceded by `$`. AT&T opcodes have suffixes such as `b` (for byte) and `w` (for word) to specify the size of memory operands, whereas Intel syntax accomplishes this by looking at the operands rather than the opcodes. To move pointer references in Intel syntax, you have to specify operand prefixes such as `byte ptr`.

> The advantage of learning AT&T syntax is that it's understood by GAS and inline GCC, which work not only on Intel-based systems, but also on a variety of processor architectures.

Next, let's rewrite the preceding snippet using GCC inline assembly, which is what you would use from the protected mode kernel:

```
unsigned short port = 0x3BC;
unsigned char data  = 0xAB;

asm("outb %%al,    %%dx\n\t"
      :
      : "a" (data), "d" (port)
   );
```

The general format of the `asm` construct supported by GCC is as follows:

```
asm(assembly
      : output operand constraints
      : input operand constraints
      : clobbered operand specifier
   );
```

In the operand sections, `a`, `b`, `c`, `d`, `S`, and `D` stand for `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, and `EDI` registers, respectively. Input operand constraints copy data from the supplied variables to the specified registers before executing the assembly instructions, whereas output operand constraints (written as `=a`, `=b`, and so on) copy data from the specified registers to the supplied variables after executing the assembly instructions. The clobbered operand constraints ask GCC to assume that the listed registers are not available for use. Look at the GCC Inline Assembly HOWTO (www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html) for more details on the GCC inline assembly syntax.

The only constraint used in our example is specific to input operands. This effectively copies the value of `data` to the `AL` register and the value of `port` to the `DX` register. Register names are preceded by `%%` in inline assembly, because `%` is used to refer to the supplied operands. `%i` stands for the i^{th} operand; so, if you want to refer to `data` and `port` inside the example inline assembly snippet, you may respectively use `%0` and `%1`.

To obtain a clearer picture of inline assembly translation, let's look at the assembly code generated by the compiler corresponding to the preceding inline assembly snippet by supplying the `-s` command-line argument to GCC. Look at the comment against each generated code line for explanations:

```
    movw $956, -2(%ebp)  # Value of data in stack set to 0x3BC
    movb $-85, -3(%ebp)  # Value of port in stack set to 0xAB
    movb -3(%ebp), %al   # movb 0xAB,  %al
    movw -2(%ebp), %dx   # movw 0x3BC, %dx
#APP                     # Marker to note start of inline assembly
    outb %al, %dx        # Write to parallel port
#NO_APP                  # Marker to note end of inline assembly
```

You may use inline assembly from user mode programs, too. Here is an application written using inline assembly that invokes the `syslog()` system call to read the last 128 bytes from the kernel `printk()` ring buffer:

Code View:

```
#define READ_COMMAND   3    /* First argument to
                               syslog() system call */
#define MSG_LENGTH     128  /* Third argument to syslog() */
```

```
int
main(int argc, char *argv[])
{
  int syslog_command = READ_COMMAND;
  int bytes_to_read  = MSG_LENGTH;
  int retval;
  char buffer[MSG_LENGTH]; /* Second argument to syslog() */

  asm volatile(
              "movl %1, %%ebx\n"     /* READ_COMMAND */
              "movl %2, %%ecx\n"     /* buffer */
              "movl %3, %%edx\n"     /* bytes_to_read */
              "movl $103, %%eax\n"   /* __NR_syslog */
              "int $128\n"           /* Generate System Call */
              "movl %%eax, %0"       /* retval */
              :"=r" (retval)
              :"m"(syslog_command),"r"(buffer),"m"(bytes_to_read)
              :"%eax","%ebx","%ecx","%edx");

  if (retval > 0) printf("%s\n", buffer);
}
```

As you learned in Chapter 4, "Laying the Groundwork," the `int $128` (or `int 0x80`) instruction generates a software interrupt that traps into system calls. Because system calls result in transition from user mode to kernel mode, the function arguments are not passed in user or kernel stacks, but in CPU registers. The system call number (*include/asm-your-arch/unistd.h* has the full list) is stored in the `EAX` register. For the `syslog()` system call, this number is 103. If you look at the man page for `syslog()`, you will see that it takes three arguments: a command, the address of a buffer to hold returned data, and the length of the buffer. These are passed in registers `EBX`, `ECX` and `EDX`, respectively. The return value is transferred from `EAX` to `retval`. The inline assembly invocation effectively translates to this:

```
retval = syslog(syslog_command, buffer, bytes_to_read);
```

If you compile and run the code, you will see output like this, fetched from the kernel ring buffer:

```
0:0:0:0: Attached scsi removable disk sda
<5>sd 0:0:0:0: Attached scsi generic sg0 type 0
<7>usb-storage: device scan complete
...
```

The kernel system call trap in *arch/x86/kernel/entry_32.S* saves all register contents to stack, so the real system calls see their arguments on stack, even though user-space code passes them in CPU registers. To ensure that system call routines expect arguments on stack, they are all tagged with the GCC attribute, `asmlinkage`. Note that `asmlinkage` has nothing to do with the `asm` (or `__asm__`) that is used to declare inline assembly.

Let's end this section by illustrating an example of inline assembly modification to a Linux bootloader for a PowerPC-based board. Assume that the flash memory on the board does not support *BackGround Operation* (BGO). This means that the bootloader code cannot write to flash while executing from flash, which is needed, for example, if the bootloader needs to update a kernel image that is residing in another part of the flash. One solution is to modify the bootloader so that the boot code used to write and erase the flash gets executed entirely from *Instruction Cache* (I-cache) with the data segment residing in *Data Cache* (D-cache). The sample

macro written here in GCC inline assembly does the job of pretouching the necessary bootloader instructions onto I-cache. You need a working knowledge of PowerPC assembly to understand this code snippet:

Code View:

```
/* instr_length is the number of instructions to touch
   into I-cache. _load_i$_copy and _end_i$_copy are
   program labels */
#define load_into_icache_copy(instr_length)        \
asm volatile("lis     %%r3, 0x1@h\n                \
              ori     %%r3, %%r3, 0x1@l\n          \
              mticcr  %%r3\n                       \
              isync\n                              \
              \n                                   \
              lis     %%r6, _end_i$_copy@h\n       \
              ori     %%r6, %%r6, _end_i$_copy@l\n \
              icbt    %%r0, %%r6\n                 \
              lis     %%r4, %0@h\n                 \
              ori     %%r4, %%r4, %0@l\n           \
              mtctr   %%r4\n                       \
      _load_i$_copy:                               \
              addis   %%r6, %%r6, 32@ha\n          \
              addi    %%r6, %%r6, 32@l\n           \
              icbt    %%r0, %%r6\n                 \
              bdnz    _load_i$_copy\n              \
      _end_i$_copy:                                \
              nop\n"                               \
      :                                            \
      : "i"(instr_length)                          \
      :"%r6","%r4","%r0","r8","r9");
```

Debugging

To debug the real mode kernel, you cannot use debuggers such as the *Kernel Debugger* (kdb) or the *Kernel GNU Debugger* (kgdb), which we discussed in Chapter 21, "Debugging Device Drivers." A quick way to debug kernel assembly snippets is by using the DOS *debug* tool after converting your code to Intel-style syntax. But *debug* was created in the 16-bit era, so you can't, for instance, step through code that initializes the EAX register. You can find 32-bit *debug*-type freeware tools available for download on the Internet. JTAG debuggers, also discussed in Chapter 21, are a kind of panacea because a single tool can be used to debug the BIOS, bootloader, Linux real mode code, and kernel-BIOS interactions.

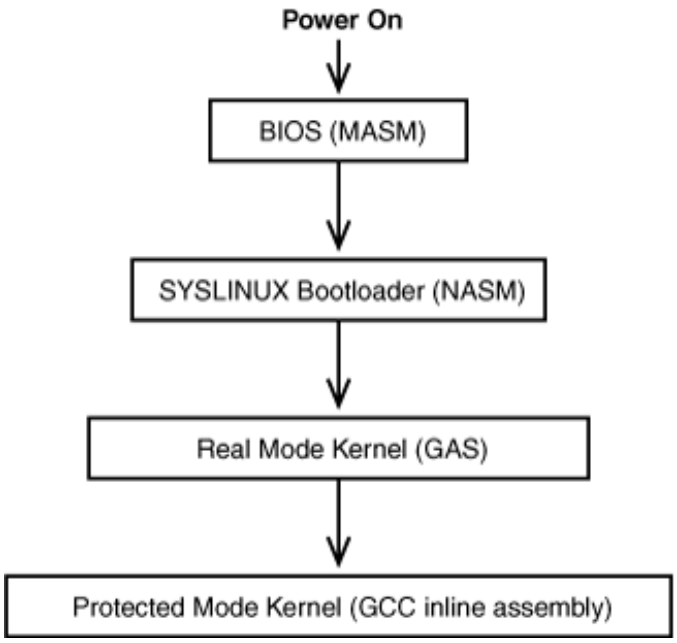
Appendix A. Linux Assembly

Device drivers sometimes need to implement some code snippets in assembly, so let's take a look at the different facets of assembly programming on Linux.

Figure A.1 shows the Linux boot sequence on a PC-compatible system and is a simpler version of Figure 2.1 in Chapter 2, "A Peek Inside the Kernel." The firmware components in the figure are implemented using different assembly syntaxes:

- The BIOS is typically written wholly in assembly. Some of the popular PC BIOSes are coded using assemblers such as the *Microsoft Macro Assembler* (MASM).
- Linux bootloaders such as LILO and GRUB are implemented using a mix of C and assembly. The SYSLINUX bootloader is entirely written in assembly using the *Netwide Assembler* (NASM).
- Real mode Linux startup code uses the *GNU Assembler* (GAS).
- Protected mode BIOS invocations are done in *inline assembly*, which is a construct supported by GCC to insert assembly in between C statements.

Figure A.1. Firmware components and assembly syntaxes.



In Figure A.1, the top two components generally follow Intel-based assembly syntax, whereas the bottom two are coded in AT&T (or GAS) syntax. There are exceptions; the assembly parts of GRUB use GAS.

To illustrate the differences between these two syntaxes, consider code that outputs a byte to the parallel port. In Intel format used by the BIOS or the bootloader, you would write the following:

```
mov dx, 03BCh   ;0x3BC is the I/O address of the parallel port
mov al, 0ABh    ;0xAB is the data to be output
out dx, al      ;Send data to the parallel port
```

However, if you want to perform the same task from Linux real mode startup code, you need to do this:

```
movw $0x3BC, %dx
movb $0xAB,  %al
outb %al,    %dx
```

You can see that unlike in Intel format, in AT&T syntax, the source operand comes first, and the destination operand comes second. Register names in AT&T format are preceded by `%`, and immediate operands are preceded by `$`. AT&T opcodes have suffixes such as `b` (for byte) and `w` (for word) to specify the size of memory operands, whereas Intel syntax accomplishes this by looking at the operands rather than the opcodes. To move pointer references in Intel syntax, you have to specify operand prefixes such as `byte ptr`.

> The advantage of learning AT&T syntax is that it's understood by GAS and inline GCC, which work not only on Intel-based systems, but also on a variety of processor architectures.

Next, let's rewrite the preceding snippet using GCC inline assembly, which is what you would use from the protected mode kernel:

```
unsigned short port = 0x3BC;
unsigned char data  = 0xAB;

asm("outb %%al,    %%dx\n\t"
      :
      : "a" (data), "d" (port)
   );
```

The general format of the `asm` construct supported by GCC is as follows:

```
asm(assembly
      : output operand constraints
      : input operand constraints
      : clobbered operand specifier
   );
```

In the operand sections, `a`, `b`, `c`, `d`, `S`, and `D` stand for `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, and `EDI` registers, respectively. Input operand constraints copy data from the supplied variables to the specified registers before executing the assembly instructions, whereas output operand constraints (written as `=a`, `=b`, and so on) copy data from the specified registers to the supplied variables after executing the assembly instructions. The clobbered operand constraints ask GCC to assume that the listed registers are not available for use. Look at the GCC Inline Assembly HOWTO (www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html) for more details on the GCC inline assembly syntax.

The only constraint used in our example is specific to input operands. This effectively copies the value of `data` to the `AL` register and the value of `port` to the `DX` register. Register names are preceded by `%%` in inline assembly, because `%` is used to refer to the supplied operands. `%i` stands for the i^{th} operand; so, if you want to refer to `data` and `port` inside the example inline assembly snippet, you may respectively use `%0` and `%1`.

To obtain a clearer picture of inline assembly translation, let's look at the assembly code generated by the compiler corresponding to the preceding inline assembly snippet by supplying the `-s` command-line argument to GCC. Look at the comment against each generated code line for explanations:

```
    movw $956, -2(%ebp)  # Value of data in stack set to 0x3BC
    movb $-85, -3(%ebp)  # Value of port in stack set to 0xAB
    movb -3(%ebp), %al   # movb 0xAB,  %al
    movw -2(%ebp), %dx   # movw 0x3BC, %dx
#APP                     # Marker to note start of inline assembly
    outb %al, %dx        # Write to parallel port
#NO_APP                  # Marker to note end of inline assembly
```

You may use inline assembly from user mode programs, too. Here is an application written using inline assembly that invokes the `syslog()` system call to read the last 128 bytes from the kernel `printk()` ring buffer:

Code View:

```
#define READ_COMMAND   3    /* First argument to
                               syslog() system call */
#define MSG_LENGTH     128  /* Third argument to syslog() */
```

```
int
main(int argc, char *argv[])
{
  int syslog_command = READ_COMMAND;
  int bytes_to_read  = MSG_LENGTH;
  int retval;
  char buffer[MSG_LENGTH]; /* Second argument to syslog() */

  asm volatile(
              "movl %1, %%ebx\n"     /* READ_COMMAND */
              "movl %2, %%ecx\n"     /* buffer */
              "movl %3, %%edx\n"     /* bytes_to_read */
              "movl $103, %%eax\n"   /* __NR_syslog */
              "int $128\n"           /* Generate System Call */
              "movl %%eax, %0"       /* retval */
              :"=r" (retval)
              :"m"(syslog_command),"r"(buffer),"m"(bytes_to_read)
              :"%eax","%ebx","%ecx","%edx");

  if (retval > 0) printf("%s\n", buffer);
}
```

As you learned in Chapter 4, "Laying the Groundwork," the `int $128` (or `int 0x80`) instruction generates a software interrupt that traps into system calls. Because system calls result in transition from user mode to kernel mode, the function arguments are not passed in user or kernel stacks, but in CPU registers. The system call number (*include/asm-your-arch/unistd.h* has the full list) is stored in the `EAX` register. For the `syslog()` system call, this number is 103. If you look at the man page for `syslog()`, you will see that it takes three arguments: a command, the address of a buffer to hold returned data, and the length of the buffer. These are passed in registers `EBX`, `ECX` and `EDX`, respectively. The return value is transferred from `EAX` to `retval`. The inline assembly invocation effectively translates to this:

```
retval = syslog(syslog_command, buffer, bytes_to_read);
```

If you compile and run the code, you will see output like this, fetched from the kernel ring buffer:

```
0:0:0:0: Attached scsi removable disk sda
<5>sd 0:0:0:0: Attached scsi generic sg0 type 0
<7>usb-storage: device scan complete
...
```

The kernel system call trap in *arch/x86/kernel/entry_32.S* saves all register contents to stack, so the real system calls see their arguments on stack, even though user-space code passes them in CPU registers. To ensure that system call routines expect arguments on stack, they are all tagged with the GCC attribute, `asmlinkage`. Note that `asmlinkage` has nothing to do with the `asm` (or `__asm__`) that is used to declare inline assembly.

Let's end this section by illustrating an example of inline assembly modification to a Linux bootloader for a PowerPC-based board. Assume that the flash memory on the board does not support *BackGround Operation* (BGO). This means that the bootloader code cannot write to flash while executing from flash, which is needed, for example, if the bootloader needs to update a kernel image that is residing in another part of the flash. One solution is to modify the bootloader so that the boot code used to write and erase the flash gets executed entirely from *Instruction Cache* (I-cache) with the data segment residing in *Data Cache* (D-cache). The sample

macro written here in GCC inline assembly does the job of pretouching the necessary bootloader instructions onto I-cache. You need a working knowledge of PowerPC assembly to understand this code snippet:

Code View:

```
/* instr_length is the number of instructions to touch
   into I-cache. _load_i$_copy and _end_i$_copy are
   program labels */
#define load_into_icache_copy(instr_length)        \
asm volatile("lis     %%r3, 0x1@h\n                \
              ori     %%r3, %%r3, 0x1@l\n          \
              mticcr  %%r3\n                       \
              isync\n                              \
              \n                                   \
              lis     %%r6, _end_i$_copy@h\n       \
              ori     %%r6, %%r6, _end_i$_copy@l\n \
              icbt    %%r0, %%r6\n                 \
              lis     %%r4, %0@h\n                 \
              ori     %%r4, %%r4, %0@l\n           \
              mtctr   %%r4\n                       \
      _load_i$_copy:                               \
              addis   %%r6, %%r6, 32@ha\n          \
              addi    %%r6, %%r6, 32@l\n           \
              icbt    %%r0, %%r6\n                 \
              bdnz    _load_i$_copy\n              \
      _end_i$_copy:                                \
              nop\n"                               \
      :                                            \
      : "i"(instr_length)                          \
      :"%r6","%r4","%r0","r8","r9");
```

Debugging

To debug the real mode kernel, you cannot use debuggers such as the *Kernel Debugger* (kdb) or the *Kernel GNU Debugger* (kgdb), which we discussed in Chapter 21, "Debugging Device Drivers." A quick way to debug kernel assembly snippets is by using the DOS *debug* tool after converting your code to Intel-style syntax. But *debug* was created in the 16-bit era, so you can't, for instance, step through code that initializes the EAX register. You can find 32-bit *debug*-type freeware tools available for download on the Internet. JTAG debuggers, also discussed in Chapter 21, are a kind of panacea because a single tool can be used to debug the BIOS, bootloader, Linux real mode code, and kernel-BIOS interactions.

Appendix B. Linux and the BIOS

Parts of the x86 kernel, such as the *video frame buffer driver* (`vesafb`) and *Advanced Power Management* (APM), explicitly use BIOS services to accomplish certain functions. Other sections of the kernel, such as the serial driver, implicitly depend on the BIOS to initialize I/O base addresses and interrupt levels. Real mode kernel code makes extensive use of BIOS calls during boot to perform tasks such as assembling the system memory map.[1] Because some device drivers depend directly or indirectly on the BIOS, let's learn how to interact with it.

[1] On BIOS-less embedded architectures, similar responsibilities (for example, waking the kernel from suspend on ARM Linux) rest with the bootloader.

Real Mode Calls

Many parts of the kernel glean information from the BIOS in real mode and use the collected information during normal operation in protected mode.

The steps needed to accomplish this are as follows:

1. Real mode kernel code invokes BIOS services and populates returned information in the first physical memory page, called the *zero page.* This is done by the source files in the *arch/x86/boot/* directory. The full layout of the zero page can be found in *Documentation/i386/zero-page.txt*.

2. After the kernel switches to protected mode, but before it clears the zero page, the obtained data is saved in kernel data structures. This is done in *arch/x86/kernel/setup_32.c*.

3. The protected mode kernel makes suitable use of the saved information during normal operation.

As an example, let's find out how the kernel assembles the system memory map from the BIOS. Listing B.1 is a snippet from *arch/x86/boot/memory.c* in the 2.6.23.1 source tree that invokes the BIOS `int 0x15` service to obtain the system memory map.

Listing B.1. Obtaining the System Memory Map (*arch/x86/boot/memory.c*)

```
static int detect_memory_e820(void)
{
  int count = 0;
  u32 next = 0;
  u32 size, id;
  u8 err;
  /* The boot_params structure contains the zero page */
      struct e820entry *desc = boot_params.e820_map;

  do {
    size = sizeof(struct e820entry);
    asm("int $0x15; setc %0"
          : "=d" (err), "+b" (next), "=a" (id), "+c" (size),
            "=m" (*desc)
          : "D" (desc), "d" (SMAP), "a" (0xe820));

    /* ... */

    count++;
    desc++;
  } while (next && count < E820MAX);

  return boot_params.e820_entries = count;
}
```

In the listing, `0xe820` is the function number specified in the `AX` register before invoking `int 0x15` to procure the memory map. If you look at the BIOS call definition for `int 0x15`, function `0xe820` (the full list is available at http://lrs.fmi.uni-passau.de/support/doc/interrupt-57/INT.HTM), you will see that the BIOS writes the current element of the memory map in a buffer pointed to by the `DI` register. In Listing B.1, `DI` points to the offset in the zero page where the memory map is to be stored (`boot_params.e820_map`). The code then loops until all elements in the memory map are collected. The number of elements is computed and stored at offset `boot_params.e820_entries` in the zero page. When execution successfully exits the loop, the memory map is available in the zero page in the form of `struct e820map`, defined in *include/asm-x86/e820.h*:

```
struct e820entry {
  _u64 addr;  /* start of memory segment */
  _u64 size;  /* size of memory segment */
  _u32 type;  /* type of memory segment */
} _attribute_((packed));

struct e820map {
  _u32 nr_map;
  struct e820entry map[E820MAX];
};
```

The kernel switches to protected mode later in *arch/x86/boot/pm.c.* When in protected mode, the kernel saves the collected memory map via `copy_e820_map()`, defined in *arch/x86/kernel/e820_32.c.* This is shown in Listing B.2. For simplicity, the listing scissors out error checks and folds the `add_memory_region()` routine.

Listing B.2. Copying the Memory Map (*arch/x86/kernel/e820_32.c*)

Code View:

```
struct e820map e820;

static int __init
copy_e820_map(struct e820entry *biosmap, int nr_map)
{
  int x;
  /* ... */

  do {
    /* Copy memory map information collected from
       the BIOS into local variables */
    unsigned long long start = biosmap->addr;
    unsigned long long size = biosmap->size;
    unsigned long long end = start + size;
    unsigned long type = biosmap->type;

    /* Sanitize start and size */
    /* ... */

    /* Populate the kernel data structure, e820 */
    x = e820.nr_map;
    e820.map[x].addr = start;
    e820.map[x].size = size;
    e820.map[x].type = type;
    e820.nr_map++;
  } while (biosmap++,--nr_map); /*Do for all elements in map*/

  /* ... */
}
```

Look at *arch/x86/mm/init_32.c* to see how the `e820` structure populated in Listing B.2 is used later on in the boot process.

The Old i386 Boot Code

Starting with the 2.6.23 kernel, the i386 boot assembly code has been largely rewritten in C. Prior to 2.6.23, the code in Listing B.1 lived in *arch/i386/boot/setup.S* rather than in *arch/x86/boot/memory.c.* Also, the switch to protected mode now occurs in *arch/x86/boot/pm.c* rather than *setup.S*.

To take another example, the kernel makes use of the BIOS `int 0x10` service to obtain video mode parameters while it's in real mode (*arch/x86/boot/video*.c*). The VESA frame buffer driver (*drivers/video/vesafb.c*) relies on these parameters to turn on graphics mode at boot time.

As an exercise, use a similar approach to obtain BIOS *Power-On Self Test* (POST) error codes from the real mode kernel (via `int 0x15`, function `0x2100`) and display them during normal operation via the */proc* filesystem.

Bootloaders also make use of BIOS services in real mode. If you browse through the sources of LILO, GRUB, or SYSLINUX, you will see a liberal sprinkling of `int 0x13` calls to read the kernel image from the boot device.

Appendix B. Linux and the BIOS

Parts of the x86 kernel, such as the *video frame buffer driver* (`vesafb`) and *Advanced Power Management* (APM), explicitly use BIOS services to accomplish certain functions. Other sections of the kernel, such as the serial driver, implicitly depend on the BIOS to initialize I/O base addresses and interrupt levels. Real mode kernel code makes extensive use of BIOS calls during boot to perform tasks such as assembling the system memory map.[1] Because some device drivers depend directly or indirectly on the BIOS, let's learn how to interact with it.

[1] On BIOS-less embedded architectures, similar responsibilities (for example, waking the kernel from suspend on ARM Linux) rest with the bootloader.

Real Mode Calls

Many parts of the kernel glean information from the BIOS in real mode and use the collected information during normal operation in protected mode.

The steps needed to accomplish this are as follows:

1. Real mode kernel code invokes BIOS services and populates returned information in the first physical memory page, called the *zero page.* This is done by the source files in the *arch/x86/boot/* directory. The full layout of the zero page can be found in *Documentation/i386/zero-page.txt*.

2. After the kernel switches to protected mode, but before it clears the zero page, the obtained data is saved in kernel data structures. This is done in *arch/x86/kernel/setup_32.c*.

3. The protected mode kernel makes suitable use of the saved information during normal operation.

As an example, let's find out how the kernel assembles the system memory map from the BIOS. Listing B.1 is a snippet from *arch/x86/boot/memory.c* in the 2.6.23.1 source tree that invokes the BIOS `int 0x15` service to obtain the system memory map.

Listing B.1. Obtaining the System Memory Map (*arch/x86/boot/memory.c*)

```
static int detect_memory_e820(void)
{
  int count = 0;
  u32 next = 0;
  u32 size, id;
  u8 err;
  /* The boot_params structure contains the zero page */
      struct e820entry *desc = boot_params.e820_map;

  do {
    size = sizeof(struct e820entry);
    asm("int $0x15; setc %0"
          : "=d" (err), "+b" (next), "=a" (id), "+c" (size),
            "=m" (*desc)
          : "D" (desc), "d" (SMAP), "a" (0xe820));

    /* ... */

    count++;
    desc++;
  } while (next && count < E820MAX);

  return boot_params.e820_entries = count;
}
```

In the listing, `0xe820` is the function number specified in the `AX` register before invoking `int 0x15` to procure the memory map. If you look at the BIOS call definition for `int 0x15`, function `0xe820` (the full list is available at http://lrs.fmi.uni-passau.de/support/doc/interrupt-57/INT.HTM), you will see that the BIOS writes the current element of the memory map in a buffer pointed to by the `DI` register. In Listing B.1, `DI` points to the offset in the zero page where the memory map is to be stored (`boot_params.e820_map`). The code then loops until all elements in the memory map are collected. The number of elements is computed and stored at offset `boot_params.e820_entries` in the zero page. When execution successfully exits the loop, the memory map is available in the zero page in the form of `struct e820map`, defined in *include/asm-x86/e820.h*:

```
struct e820entry {
  _u64 addr;  /* start of memory segment */
  _u64 size;  /* size of memory segment */
  _u32 type;  /* type of memory segment */
} _attribute_((packed));

struct e820map {
  _u32 nr_map;
  struct e820entry map[E820MAX];
};
```

The kernel switches to protected mode later in *arch/x86/boot/pm.c.* When in protected mode, the kernel saves the collected memory map via `copy_e820_map()`, defined in *arch/x86/kernel/e820_32.c.* This is shown in Listing B.2. For simplicity, the listing scissors out error checks and folds the `add_memory_region()` routine.

Listing B.2. Copying the Memory Map (*arch/x86/kernel/e820_32.c*)

Code View:

```
struct e820map e820;

static int __init
copy_e820_map(struct e820entry *biosmap, int nr_map)
{
  int x;
  /* ... */

  do {
    /* Copy memory map information collected from
       the BIOS into local variables */
    unsigned long long start = biosmap->addr;
    unsigned long long size = biosmap->size;
    unsigned long long end = start + size;
    unsigned long type = biosmap->type;

    /* Sanitize start and size */
    /* ... */

    /* Populate the kernel data structure, e820 */
    x = e820.nr_map;
    e820.map[x].addr = start;
    e820.map[x].size = size;
    e820.map[x].type = type;
    e820.nr_map++;
  } while (biosmap++,--nr_map); /*Do for all elements in map*/

  /* ... */
}
```

Look at *arch/x86/mm/init_32.c* to see how the `e820` structure populated in Listing B.2 is used later on in the boot process.

The Old i386 Boot Code

Starting with the 2.6.23 kernel, the i386 boot assembly code has been largely rewritten in C. Prior to 2.6.23, the code in Listing B.1 lived in *arch/i386/boot/setup.S* rather than in *arch/x86/boot/memory.c.* Also, the switch to protected mode now occurs in *arch/x86/boot/pm.c* rather than *setup.S*.

To take another example, the kernel makes use of the BIOS `int 0x10` service to obtain video mode parameters while it's in real mode (*arch/x86/boot/video*.c*). The VESA frame buffer driver (*drivers/video/vesafb.c*) relies on these parameters to turn on graphics mode at boot time.

As an exercise, use a similar approach to obtain BIOS *Power-On Self Test* (POST) error codes from the real mode kernel (via `int 0x15`, function `0x2100`) and display them during normal operation via the */proc* filesystem.

Bootloaders also make use of BIOS services in real mode. If you browse through the sources of LILO, GRUB, or SYSLINUX, you will see a liberal sprinkling of `int 0x13` calls to read the kernel image from the boot device.

Protected Mode Calls

To see how the kernel makes protected mode BIOS calls, let's look at the APM implementation.

APM is a BIOS interface specification, which is now almost obsolete (see the section "Power Management" in Chapter 4, "Laying the Groundwork"). Power management policies are defined in the BIOS, and a kernel thread called *kapmd* polls it every second to figure out the course of action. The polling is done using protected mode BIOS calls. To do this, *kapmd* needs to know the protected mode entry segment address and offset. These are obtained from the real mode kernel during boot using the `int 0x15`, function `0x5303` BIOS service.

The actual protected mode BIOS call is invoked using inline assembly from `apm_bios_call_simple_asm()`, defined in *include/asm-x86/mach-default/apm.h*:

```
__asm__ __volatile__(APM_DO_ZERO_SEGS
    "pushl %%edi\n\t"
    "pushl %%ebp\n\t"
    "lcall *%%cs:apm_bios_entry\n\t"
    "setc %%bl\n\t"
    "popl %%ebp\n\t"
    "popl %%edi\n\t"
    APM_DO_POP_SEGS
        : "=a" (*eax), "=b" (error), "=c" (cx), "=d" (dx),
          "=S" (si)
        : "a" (func), "b" (ebx_in), "c" (ecx_in)
        : "memory", "cc");
```

`APM_DO_ZERO_SEGS` zeros out segment registers. `apm_bios_entry` contains the protected mode entry address. The input constraint `"a"(func)` copies the desired BIOS function number to the `EAX` register before invocation. For example, function number `APM_FUNC_GET_EVENT` (`0x530b`) elicits an APM event from the BIOS, and function number `APM_FUNC_IDLE` (`0x5305`) notifies the BIOS that the processor is idle. Results are returned by the BIOS in registers `EAX`, `EBX`, `ECX`, and `EDX`. As per the previous output operand constraints, these are propagated to the caller in variables `*eax`, `error`, `cx`, and `dx`, respectively. In the assembly body, registers are saved onto the kernel stack before the BIOS call and restored afterward to prevent the BIOS from trampling on them.

BIOS and Legacy Drivers

The BIOS provides a degree of hardware abstraction to some Linux drivers. Let's take the PC serial port driver (discussed in Chapter 6, "Serial Drivers") as an example. The BIOS probes the Super I/O chipset and assigns I/O base addresses and IRQs to the respective serial (and Infrared) ports. The serial driver needs to be told about the resources assigned by the BIOS either via hard-coded values in a header file (*include/asm-x86/serial.h*) or via user-space commands. As an exercise, dig into the data sheet of your Super I/O chipset and add support in the serial driver to probe for the resource values set by the BIOS.

To take another example, even if you disable USB support in the kernel, you can use USB keyboards and mice on PC systems with help from the BIOS. The BIOS turns on an emulation mode in the South Bridge that routes USB keyboard and mouse input from the USB controller to the keyboard controller. This tricks the operating system into thinking that you are using a legacy keyboard or mouse.

The kernel used to rely on the BIOS to walk the PCI bus and configure detected devices. This is now obsolete, but take a look at *arch/x86/pci/pcbios.c* to see how PCI BIOS can be accessed from the kernel. Chapter 10, "Peripheral Component Interconnect," discussed PCI drivers.

Appendix C. Seq Files

Monitoring and trending data points offered by procfs might help diagnose device driver problems when the cause of a symptom looks fuzzy. But sometimes, especially when the amount of data is large, the corresponding procfs `read()` implementations become complex. The seq file interface is a kernel helper mechanism designed to simplify such implementations. Seq files render procfs operations cleaner and easier.

Let's gradually introduce complexities to a procfs `read()` routine and see how the seq file interface transforms the labored routine into a graceful one. We'll also update one of the few remaining 2.6 drivers that does not yet leverage seq files.

The Seq File Advantage

Let's discover the advantages offered by seq files with the help of an example. As is common with many device drivers, assume that you have a linked list of data structures and that each node in the list contains a string field (called `info`). The example code in Listing C.1 uses a procfs file named */proc/readme* to export these strings to user space. When a user reads this file, the procfs `read()` method, `readme_proc()`, gets invoked. This routine traverses the linked list and appends the `info` field of each node to the filesystem buffer passed down to it.

Listing C.1. Reading via Procfs

Code View:

```
/* Private Data structure */
struct _mydrv_struct {
  /* ... */
  struct list_head list;  /* Link to the next node */
  char info[10];          /* Info to pass via the procfs file */
  /* ... */
};

static LIST_HEAD(mydrv_list);  /* List Head */

/* Initialization */
static int __init
mydrv_init(void)
{
  int i;
  static struct proc_dir_entry *entry = NULL ;
  struct _mydrv_struct *mydrv_new;

  /* ... */
  /* Create /proc/readme */
  entry = create_proc_entry("readme", S_IWUSR, NULL);

  /* Attach it to readme_proc() */
  if (entry) {
    entry->read_proc = readme_proc;
  }
```

```
  /* Handcraft mydrv_list for testing purpose.
     In the real world, device driver logic
     maintains the list and populates the 'info' field */
  for (i=0;i<100;i++) {
    mydrv_new = kmalloc(sizeof(mydrv_struct), GFP_ATOMIC);
    sprintf(mydrv_new->info, "Node No: %d\n", i);
    list_add_tail(&mydrv_new->list, &mydrv_list);
  }
  return 0;
}

/* The procfs read entry point */
static int
readme_proc(char *page, char **start, off_t offset,
            int count, int *eof, void *data)
{
  int i = 0;
  off_t thischunk_len = 0;
  struct _mydrv_struct *p;

  /* Traverse the list and copy info into the supplied buffer */
  list_for_each_entry(p, &mydrv_list, list) {
    thischunk_len += sprintf(page+thischunk_len, p->info);
  }
  *eof = 1; /* Indicate completion */
  return thischunk_len;
}
```

Boot the kernel with these changes and peek inside */proc/readme:*

```
bash> cat /proc/readme
Node No: 0
Node No: 1
...
Node No: 99
```

When procfs `read()` methods are invoked, they are supplied one page of memory that they can use to pass information to user space. As you can see in Listing C.1, the first argument passed to `readme_proc()` is a pointer to this page-sized buffer. The second argument, `start`, is used to aid the implementation of procfs files larger than a page. The use of this parameter will get clear when we look at the example in Listing C.2. The next two arguments respectively specify the offset from where the read operation is requested and the number of bytes to be read. The `eof` argument is used to tell the caller whether there is more data to be read. If `*eof` is not set before returning, the procfs read entry point is called again for more data. In Listing C.1, if you comment out the line that sets `*eof`, `readme_proc()` gets called again with the offset argument set to 1190 (which is the number of ASCII bytes contained in the strings, Node No: 0 to Node No: 99). `readme_proc()` returns the number of bytes copied to the supplied buffer.

The size of data generated by the procfs read routine in Listing C.1 falls within the one-page limit. However, if you increase the number of nodes in the linked list from 100 to 500 in `mydrv_init()`, the amount of data generated while reading */proc/readme* crosses a page and triggers the following output:

```
bash> cat /proc/readme
Node No: 0
```

```
Node No: 1
...
Node No: 322
proc_file_read: Apparent buffer overflow!
```

As you can see, an overflow occurs after one page (4,096 in this case) worth of ASCII characters have been produced.

To handle such large procfs files, you need to refashion the code in Listing C.1 using the `start` parameter alluded to earlier. This makes the function somewhat complicated and is shown in Listing C.2. The semantics of this modified implementation is as follows:

- `readme_proc()` is called multiple times, each invocation yielding a maximum of `count` bytes starting at `offset`. The `count` requested during each call is less than the size of a page.
- During each invocation, the kernel increments `offset` by the number of bytes returned by the previous invocation.
- `readme_proc()` signals `eof` only if the amount of data produced is less than or equal to the requested `count` plus the current `offset`. If `eof` is not set, the function is called again with `offset` advanced by the number of bytes returned previously.
- After each invocation, only those bytes starting from `*start` are collected and returned to the caller.

Print the values of `*start`, `offset`, `count`, and `page`, and look at the output generated during each invocation to better understand the operation sequence.

With this hack, your procfs file can supply large amounts of data to user space without size limitations:

```
bash> cat /proc/readme
Node No: 0
Node No: 1
...
Node No: 499
```

Listing C.2. Large Procfs Reads

Code View:

```
static int
readme_proc(char *page, char **start, off_t offset,
               int count, int *eof, void *data)

{
  int i = 0;
  off_t thischunk_start = 0;
  off_t thischunk_len = 0;
  struct _mydrv_struct *p;
  /* Loop thru the list collecting device info */
  list_for_each_entry(p, &mydrv_list, list) {
    thischunk_len += sprintf(page+thischunk_len, p->info);

    /* Advance thischunk_start only to the extent that the next
     * read will not result in total bytes more than (offset+count)
     */
    if (thischunk_start + thischunk_len < offset) {
      thischunk_start += thischunk_len;
      thischunk_len = 0;
    } else if (thischunk_start + thischunk_len > offset+count) {
      break;
    } else {
      continue;
    }
  }

  /* Actual start */
  *start = page + (offset - thischunk_start);

  /* Calculate number of written bytes */
  thischunk_len -= (offset - thischunk_start);
  if (thischunk_len > count) {
    thischunk_len = count;
  } else {
    *eof = 1;
  }

  return thischunk_len;
}
```

The seq file interface comes to the rescue when you are faced with the prospect of awkwardly implementing large procfs files as in Listing C.2. As the name implies, the seq file interface views the contents of procfs files as a sequence of objects. Programming interfaces are provided to iterate through this object sequence. Your code has to supply the following *iterator* methods expected by the seq interface:

1. `start()`, which is called first by the seq interface. This initializes the position within the iterator sequence and returns the first iterator object of interest.

2. `next()`, which increments the iterator position and returns a pointer to the next iterator. This function is agnostic to the internal structure of the iterator and considers it an opaque object.

3. `show()`, which interprets the iterator passed to it and generates output strings to be displayed when a user reads the corresponding procfs file. This method makes use of helpers such as `seq_printf()`, `seq_putc()`, and `seq_puts()` to format the output.

4. `stop()`, which is called at the end for cleanup.

The seq file interface automatically invokes these iterator methods to produce output in response to user operations on related procfs files. You no longer need to worry about page-sized buffers and signaling the end of data.

Let's rewrite Listing C.2 making use of seq files. This is done in Listing C.3 by viewing the linked list as a sequence of nodes. The basic iterator object is the node, and each invocation of the `next()` method returns the next node in the list.

Listing C.3. Using Seq Files to Simplify Listing C.2

Code View:

```
#include <linux/seq_file.h>

/* start() method */
static void *
mydrv_seq_start(struct seq_file *seq, loff_t *pos)
{
  struct _mydrv_struct *p;
  loff_t off = 0;

  /* The iterator at the requested offset */
  list_for_each_entry(p, &mydrv_list, list) {
    if (*pos == off++) return p;
  }
  return NULL;
}

/* next() method */
static void *
mydrv_seq_next(struct seq_file *seq, void *v, loff_t *pos)
{
  /* 'v' is a pointer to the iterator returned by start() or
     by the previous invocation of next() */
  struct list_head *n = ((struct _mydrv_struct *)v)->list.next;

  ++*pos; /* Advance position */
  /* Return the next iterator, which is the next node in the list */
  return(n != &mydrv_list) ?
        list_entry(n, struct _mydrv_struct, list) : NULL;
}

/* show() method */
static int
mydrv_seq_show(struct seq_file *seq, void *v)
{
  const struct _mydrv_struct *p = v;

  /* Interpret the iterator, 'v' */
  seq_printf(seq, p->info);
```

```
  return 0;
}

/* stop() method */
static void
mydrv_seq_stop(struct seq_file *seq, void *v)
{
  /* No cleanup needed in this example */
}

/* Define iterator operations */
static struct seq_operations mydrv_seq_ops = {
  .start = mydrv_seq_start,
  .next  = mydrv_seq_next,
  .stop  = mydrv_seq_stop,
  .show  = mydrv_seq_show,
};

static int
mydrv_seq_open(struct inode *inode, struct file *file)
{
  /* Register the operators */
  return seq_open(file, &mydrv_seq_ops);
}

static struct file_operations mydrv_proc_fops = {
  .owner   = THIS_MODULE,
  .open    = mydrv_seq_open, /* User supplied */
  .read    = seq_read,       /* Built-in helper function */
  .llseek  = seq_lseek,      /* Built-in helper function */
  .release = seq_release,    /* Built-in helper funciton */
};

static int __init
mydrv_init(void)
{
  /* ... */

  /* Replace the assignment to entry->read_proc in Listing C.1,
     with a more fundamental assignment to entry->proc_fops. So
     instead of doing "entry->read_proc = readme_proc;", do the
     following: */
  entry->proc_fops = &mydrv_proc_fops;

  /* ... */
}
```

Appendix C. Seq Files

Monitoring and trending data points offered by procfs might help diagnose device driver problems when the cause of a symptom looks fuzzy. But sometimes, especially when the amount of data is large, the corresponding procfs `read()` implementations become complex. The seq file interface is a kernel helper mechanism designed to simplify such implementations. Seq files render procfs operations cleaner and easier.

Let's gradually introduce complexities to a procfs `read()` routine and see how the seq file interface transforms the labored routine into a graceful one. We'll also update one of the few remaining 2.6 drivers that does not yet leverage seq files.

The Seq File Advantage

Let's discover the advantages offered by seq files with the help of an example. As is common with many device drivers, assume that you have a linked list of data structures and that each node in the list contains a string field (called `info`). The example code in Listing C.1 uses a procfs file named */proc/readme* to export these strings to user space. When a user reads this file, the procfs `read()` method, `readme_proc()`, gets invoked. This routine traverses the linked list and appends the `info` field of each node to the filesystem buffer passed down to it.

Listing C.1. Reading via Procfs

Code View:

```
/* Private Data structure */
struct _mydrv_struct {
  /* ... */
  struct list_head list;  /* Link to the next node */
  char info[10];          /* Info to pass via the procfs file */
  /* ... */
};

static LIST_HEAD(mydrv_list);  /* List Head */

/* Initialization */
static int __init
mydrv_init(void)
{
  int i;
  static struct proc_dir_entry *entry = NULL ;
  struct _mydrv_struct *mydrv_new;

  /* ... */
  /* Create /proc/readme */
  entry = create_proc_entry("readme", S_IWUSR, NULL);

  /* Attach it to readme_proc() */
  if (entry) {
    entry->read_proc = readme_proc;
  }
```

```
  /* Handcraft mydrv_list for testing purpose.
     In the real world, device driver logic
     maintains the list and populates the 'info' field */
  for (i=0;i<100;i++) {
    mydrv_new = kmalloc(sizeof(mydrv_struct), GFP_ATOMIC);
    sprintf(mydrv_new->info, "Node No: %d\n", i);
    list_add_tail(&mydrv_new->list, &mydrv_list);
  }
  return 0;
}

/* The procfs read entry point */
static int
readme_proc(char *page, char **start, off_t offset,
            int count, int *eof, void *data)
{
  int i = 0;
  off_t thischunk_len = 0;
  struct _mydrv_struct *p;

  /* Traverse the list and copy info into the supplied buffer */
  list_for_each_entry(p, &mydrv_list, list) {
    thischunk_len += sprintf(page+thischunk_len, p->info);
  }
  *eof = 1; /* Indicate completion */
  return thischunk_len;
}
```

Boot the kernel with these changes and peek inside */proc/readme:*

```
bash> cat /proc/readme
Node No: 0
Node No: 1
...
Node No: 99
```

When procfs `read()` methods are invoked, they are supplied one page of memory that they can use to pass information to user space. As you can see in Listing C.1, the first argument passed to `readme_proc()` is a pointer to this page-sized buffer. The second argument, `start`, is used to aid the implementation of procfs files larger than a page. The use of this parameter will get clear when we look at the example in Listing C.2. The next two arguments respectively specify the offset from where the read operation is requested and the number of bytes to be read. The `eof` argument is used to tell the caller whether there is more data to be read. If `*eof` is not set before returning, the procfs read entry point is called again for more data. In Listing C.1, if you comment out the line that sets `*eof`, `readme_proc()` gets called again with the offset argument set to 1190 (which is the number of ASCII bytes contained in the strings, Node No: 0 to Node No: 99). `readme_proc()` returns the number of bytes copied to the supplied buffer.

The size of data generated by the procfs read routine in Listing C.1 falls within the one-page limit. However, if you increase the number of nodes in the linked list from 100 to 500 in `mydrv_init()`, the amount of data generated while reading */proc/readme* crosses a page and triggers the following output:

```
bash> cat /proc/readme
Node No: 0
```

```
Node No: 1
...
Node No: 322
proc_file_read: Apparent buffer overflow!
```

As you can see, an overflow occurs after one page (4,096 in this case) worth of ASCII characters have been produced.

To handle such large procfs files, you need to refashion the code in Listing C.1 using the `start` parameter alluded to earlier. This makes the function somewhat complicated and is shown in Listing C.2. The semantics of this modified implementation is as follows:

- `readme_proc()` is called multiple times, each invocation yielding a maximum of `count` bytes starting at `offset`. The `count` requested during each call is less than the size of a page.
- During each invocation, the kernel increments `offset` by the number of bytes returned by the previous invocation.
- `readme_proc()` signals `eof` only if the amount of data produced is less than or equal to the requested `count` plus the current `offset`. If `eof` is not set, the function is called again with `offset` advanced by the number of bytes returned previously.
- After each invocation, only those bytes starting from `*start` are collected and returned to the caller.

Print the values of `*start`, `offset`, `count`, and `page`, and look at the output generated during each invocation to better understand the operation sequence.

With this hack, your procfs file can supply large amounts of data to user space without size limitations:

```
bash> cat /proc/readme
Node No: 0
Node No: 1
...
Node No: 499
```

Listing C.2. Large Procfs Reads

Code View:

```
static int
readme_proc(char *page, char **start, off_t offset,
               int count, int *eof, void *data)

{
  int i = 0;
  off_t thischunk_start = 0;
  off_t thischunk_len = 0;
  struct _mydrv_struct *p;
  /* Loop thru the list collecting device info */
  list_for_each_entry(p, &mydrv_list, list) {
    thischunk_len += sprintf(page+thischunk_len, p->info);

    /* Advance thischunk_start only to the extent that the next
     * read will not result in total bytes more than (offset+count)
     */
    if (thischunk_start + thischunk_len < offset) {
      thischunk_start += thischunk_len;
      thischunk_len = 0;
    } else if (thischunk_start + thischunk_len > offset+count) {
      break;
    } else {
      continue;
    }
  }

  /* Actual start */
  *start = page + (offset - thischunk_start);

  /* Calculate number of written bytes */
  thischunk_len -= (offset - thischunk_start);
  if (thischunk_len > count) {
    thischunk_len = count;
  } else {
    *eof = 1;
  }

  return thischunk_len;
}
```

The seq file interface comes to the rescue when you are faced with the prospect of awkwardly implementing large procfs files as in Listing C.2. As the name implies, the seq file interface views the contents of procfs files as a sequence of objects. Programming interfaces are provided to iterate through this object sequence. Your code has to supply the following *iterator* methods expected by the seq interface:

1. `start()`, which is called first by the seq interface. This initializes the position within the iterator sequence and returns the first iterator object of interest.

2. `next()`, which increments the iterator position and returns a pointer to the next iterator. This function is agnostic to the internal structure of the iterator and considers it an opaque object.

3. `show()`, which interprets the iterator passed to it and generates output strings to be displayed when a user reads the corresponding procfs file. This method makes use of helpers such as `seq_printf()`, `seq_putc()`, and `seq_puts()` to format the output.

4. `stop()`, which is called at the end for cleanup.

The seq file interface automatically invokes these iterator methods to produce output in response to user operations on related procfs files. You no longer need to worry about page-sized buffers and signaling the end of data.

Let's rewrite Listing C.2 making use of seq files. This is done in Listing C.3 by viewing the linked list as a sequence of nodes. The basic iterator object is the node, and each invocation of the `next()` method returns the next node in the list.

Listing C.3. Using Seq Files to Simplify Listing C.2

Code View:

```
#include <linux/seq_file.h>

/* start() method */
static void *
mydrv_seq_start(struct seq_file *seq, loff_t *pos)
{
  struct _mydrv_struct *p;
  loff_t off = 0;

  /* The iterator at the requested offset */
  list_for_each_entry(p, &mydrv_list, list) {
    if (*pos == off++) return p;
  }
  return NULL;
}

/* next() method */
static void *
mydrv_seq_next(struct seq_file *seq, void *v, loff_t *pos)
{
  /* 'v' is a pointer to the iterator returned by start() or
     by the previous invocation of next() */
  struct list_head *n = ((struct _mydrv_struct *)v)->list.next;

  ++*pos; /* Advance position */
  /* Return the next iterator, which is the next node in the list */
  return(n != &mydrv_list) ?
        list_entry(n, struct _mydrv_struct, list) : NULL;
}

/* show() method */
static int
mydrv_seq_show(struct seq_file *seq, void *v)
{
  const struct _mydrv_struct *p = v;

  /* Interpret the iterator, 'v' */
  seq_printf(seq, p->info);
```

```
  return 0;
}

/* stop() method */
static void
mydrv_seq_stop(struct seq_file *seq, void *v)
{
  /* No cleanup needed in this example */
}

/* Define iterator operations */
static struct seq_operations mydrv_seq_ops = {
  .start = mydrv_seq_start,
  .next  = mydrv_seq_next,
  .stop  = mydrv_seq_stop,
  .show  = mydrv_seq_show,
};

static int
mydrv_seq_open(struct inode *inode, struct file *file)
{
  /* Register the operators */
  return seq_open(file, &mydrv_seq_ops);
}

static struct file_operations mydrv_proc_fops = {
  .owner   = THIS_MODULE,
  .open    = mydrv_seq_open, /* User supplied */
  .read    = seq_read,       /* Built-in helper function */
  .llseek  = seq_lseek,      /* Built-in helper function */
  .release = seq_release,    /* Built-in helper funciton */
};

static int __init
mydrv_init(void)
{
  /* ... */

  /* Replace the assignment to entry->read_proc in Listing C.1,
     with a more fundamental assignment to entry->proc_fops. So
     instead of doing "entry->read_proc = readme_proc;", do the
     following: */
  entry->proc_fops = &mydrv_proc_fops;

  /* ... */
}
```

Updating the NVRAM Driver

The seq file interface has been around since the latter versions of the 2.4 kernel, but its use has become widespread only with 2.6. Let's update the NVRAM driver (*drivers/char/nvram.c*), one of the few remaining drivers that hasn't switched over to use seq files. (As usual, **+** and **-** show the differences from the original source file.) To do this, you may use an extra-simple flavor of seq files that uses only the `show()` iterator method. Use `single_open()` to register this method.

Listing C.4 contains the updated NVRAM driver. Because the seq interface won't sleep between calls to iterator methods, you may hold locks inside the methods.

Listing C.4. Update the NVRAM Driver Using Seq Files

Code View:

```
+static struct file_operations nvram_proc_fops = {
+  .owner   = THIS_MODULE,
+  .open    = nvram_seq_open,
+  .read    = seq_read,
+  .llseek  = seq_lseek,
+  .release = single_release,
+};
-static struct file_operations nvram_fops = {
-  .owner    = THIS_MODULE,
-  .llseek   = nvram_llseek,
-  .read     = nvram_read,
-  .write    = nvram_write,
-  .ioctl    = nvram_ioctl,
-  .open     = nvram_open,
-  .release  = nvram_release,
-};

+static int nvram_seq_open(struct inode *inode, struct file *file)
+{
+  return single_open(file, nvram_show, NULL);
+}

+static int nvram_show(struct seq_file *seq, void *v)
+{
+  unsigned char contents[NVRAM_BYTES];
+  int i;
+
+  spin_lock_irq(&rtc_lock);
+  for (i = 0; i < NVRAM_BYTES; ++i)
+  contents[i] = __nvram_read_byte(i);
+  spin_unlock_irq(&rtc_lock);
+
+  mach_proc_infos(seq, contents);
+  return 0;
+}

static int __init
nvram_init(void)
{

+  ent = create_proc_entry("driver/nvram", 0, NULL);
```

```
+  if (!ent) {
+    printk(KERN_ERR "nvram: can't create /proc/driver/nvram\n");
+    ret = -ENOMEM;
+    goto outmisc;
+  }
+  ent->proc_fops = &nvram_proc_fops;
-  if (!create_proc_read_entry("driver/nvram", 0, NULL,
-                              nvram_read_proc, NULL)) {
-    printk(KERN_ERR "nvram: can't create /proc/driver/nvram\n");
-    ret = -ENOMEM;
-    goto outmisc;
-  }
   /* ... */
}

-#define PRINT_PROC(fmt,args...)      \
-/* ... */

-static int
-nvram_read_proc(char *buffer, char **start, off_t offset,
-    int size, int *eof, void *data)
-{
- /* ... */
-}
```

In addition to the modifications in Listing C.4, change all references to `PRINT_PROC()` in the original driver to `seq_printf()`. The original driver and the one in Listing C.4 produce the same output if you read from */proc/driver/nvram*.

Looking at the Sources

Look at *Documentation/filesystems/proc.txt* for more information about procfs. The *fs/proc/* directory contains code that implements the procfs core. The seq file interface lives in *fs/seq_file.c.* Users of procfs and seq files are sprinkled all over the kernel sources.

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

$ (dollar sign)
% (percent sign)
1-wire protocol
4G networking
7-bit addressing
802.11 stack
855GME EDAC driver
8250.c driver
16550-type UART

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

AAL (ATM Adaptation Layer)
AC'97
ac97_bus module
accelerated methods
accelerometers
accessing
  char drivers
  EEPROM device
  I/O regions
  memory regions from user space
  PCI regions
    configuration space
    I/O and memory regions
  registers
access point names (APNs)
Acclerated Graphics Port (AGP)
ACPI (Advanced Configuration and Power Interface) 2nd
  acpid daemon
  AML (ACPI Machine Language Interpreter)
  devices
  drivers
  kacpid
  spaces
  user-space tools
acpid daemon
acpitool command
activation
  net_device structure
  NICs (network interface cards)
active queues
ad-hoc mode (WLAN)
ADC (Analog-to-Digital Converter) 2nd
add_disk() function 2nd
add_memory_region() function
add_mtd_partitions() function
add-symbol-file command
add_timer() function 2nd
add_wait_queue() function 2nd
addresses
  ARP (Address Resolution Protocol)
  bus addresses
  endpoint addresses
  LBA (logical block addressing)
  logical addresses
  MAC (Media Access Control) addresses
  PCI
  slave addresses
  USB (universal serial bus)
  virtual addresses
Address Resolution Protocol (ARP)
adjust checksum command (ioctl)
adjust_cmos_crc() function
Advanced Configuration and Power Interface [See ACPI (Advanced Configuration and Power Interface).]
Advanced Host Controller Interface (AHCI)
Advanced Linux Sound Architecture [See ALSA (Advanced Linux Sound Architecture).]

Advanced Power Management (APM) 2nd [See also BIOS (basic input/output system).]
Advanced Technology Attachment (ATA)
AF_INET protocol family
AF_NETLINK protocol family
AF_UNIX protocol family
Affix
AGP (Acclerated Graphics Port)
AHCI (Advanced Host Controller Interface)
AIO (Asynchronous I/O)
aio_read() function
aio_write() function
alloc_chrdev_region() function 2nd 3rd
alloc_disk() function 2nd
alloc_etherdev() function 2nd
alloc_ieee80211() function 2nd
alloc_irdadev() function 2nd
alloc_netdev() function 2nd
allocating memory
allow_signal() function 2nd
ALSA (Advanced Linux Sound Architecture)
    ALSA driver for MP3 player
    ALSA programming
alsa-devel mailing list
alsa-lib library
alsa-utils package
alsactl command
alsamixer command
amateur radio
amd_flash_info structure
amixer command
AML (ACPI Machine Language Interpreter)
Analog-to-Digital Converter (ADC) 2nd
anticipatory I/O scheduler 2nd
aplay command
APM (Advanced Power Management) 2nd [See also BIOS (basic input/output system).]
apm_bios_call_simple_asm() function
APM_DO_ZERO_SEGS
APM_FUNC_GET_EVENT
APM_FUNC_IDLE
APNs (access point names)
applying patches
arch directory
    arch/x86/boot/ directory
    arch/x86/boot/memory.c file
    arch/x86/kernel/e820_32.c file
ARM bootloaders
ARP (Address Resolution Protocol)
asked_to_die() function
asm construct
asmlinkage attribute
assembly
    boot sequence
    debugging
    GNU Assembler (GAS)
    i386 boot assembly code
    inline assembly
    Microsoft Macro Assembler (MASM)
    Netwide Assembler (NASM)
assigning IRQs (interrupt requests)
asynchronous DMA
Asynchronous I/O (AIO)
asynchronous interrupts
asynchronous transfer mode (ATM)
ATA (Advanced Technology Attachment)

ATAGs
ATAPI (ATA Packet Interface)
ATM (asynchronous transfer mode)
ATM Adaptation Layer (AAL)
atomic_dec() function
atomic_dec_and_test() function
atomic_inc() function
atomic_inc_and_test() function
atomic_notifier_chain_register() function 2nd
ATOMIC_NOTIFIER_HEAD() macro 2nd
atomic operators
Attribute memory (PCMCIA)
audio codecs
audio drivers
ALSA (Advanced Linux Sound Architecture)
ALSA driver for MP3 player
ALSA programming
audio architecture
audio codecs
Bluetooth
data structures
debugging
embedded drivers
kernel programming interfaces, table of
MP3 player example
ALSA driver code listing
ALSA programming
codec_write_reg() function
MP3 decoding complexity
mycard_audio_probe() function
mycard_audio_remove() functions
mycard_hw_params() function
mycard_pb_trigger() function
mycard_playback_open() function
overview
register layout of audio hardware
snd_card_free() function
snd_card_new() function
snd_card_proc_new() function
snd_card_register() function
snd_ctl_add() function
snd_ctl_new1() function
snd_device_new() function
snd_kcontrol structure
snd_pcm_hardware structure
snd_pcm_lib_malloc_pages() function
snd_pcm_lib_preallocate_pages_for_all() function
snd_pcm_new() function
snd_pcm_ops structure
snd_pcm_set_ops() function
user programs
OSS (Open Sound System)
overview
sound directory
sound mixing (fn)
sources
audio players [See MP3 player example.]
autoloading modules
AX.25 protocol

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

BackGround Operation (BGO)
backlight_device_register()
barriers (memory)
BCD (Binary Coded Decimal)
BCD2BIN() macro
BCSP (BlueCore Serial Protocol)
bdflush kernel thread
benchmarking
BGO (BackGround Operation)
BH (bottom half) flavors
Binary Coded Decimal (BCD)
Binutils
bio_for_each_segment() function 2nd
bio structure 2nd
bio_vec structure
BIOS (basic input/output system)
  BIOS-provided physical RAM map
  legacy drivers
  protected mode calls
  real mode calls
  updating
bit-banging drivers
blk_cleanup_queue() function
blk_fs_request() function
blk_init_queue() function 2nd
blk_queue_hardsect_size() function 2nd
blk_queue_make_request() function 2nd
blk_queue_max_sectors() function 2nd
blk_rq_map_sg() function 2nd
BLOBs (BootLoader Objects)
block device emulation
block directory
block drivers
  block_device_operations structure
  block I/O layer
  data structures 2nd
  debugging
  DMA data transfer
  entry points
  interrupt handlers
  I/O schedulers
  kernel programming interfaces, table of
  myblkdev storage controller
    block device operations
    disk access
    initialization
    overview
    register layout
  sources
  storage technologies
    ATAPI (ATA Packet Interface)
    IDE (Integrated Drive Electronics)
    libATA
    MMC (MultiMediaCard)
    RAID (redundant array of inexpensive disks)

SATA (Serial ATA)
SCSI (Small Computer System Interface)
SD (Secure Digital) cards
summary of
block I/O layer
blocking_notifier_call_chain() function 2nd
blocking_notifier_chain_register() function
BLOCKING_NOTIFIER_HEAD() macro 2nd
blocks
BlueCore Serial Protocol (BCSP)
Bluetooth 2nd
audio
Bluetooth Host Control Interface
Bluetooth Network Encapsulation Protocol (BNEP)
Bluetooth Special Interest Group (SIG)
BlueZ
CF cards
RFCOMM
USB adapters
debugging
keyboards
mice
networking
profiles
USB
bluetooth.ko
Bluetooth Host Control Interface
Bluetooth Network Encapsulation Protocol (BNEP)
Bluetooth Special Interest Group (SIG)
BlueZ
CF cards
RFCOMM
USB adapters
bluez-utils package
BNEP (Bluetooth Network Encapsulation Protocol)
bnep.ko
board rework
BogoMIPS
BootLoader Objects (BLOBs)
bootloaders
definition
embedded bootloaders
BLOB (BootLoader Object)
bootstrapping
GRUB
LILO (Linux Loader)
overview
RedBoot
SYSLINUX
table of
Redboot bootloader
boot logo (console drivers)
boot process 2nd [See also bootloaders.]
BIOS-provided physical RAM map
delay-loop calibration
EXT3 filesystem
HLT instruction
I/O scheduler
init process
initrd memory
kernel command line
Linux boot sequence
low memory/high memory
PCI resource configuration

registered protocol families
start_kernel() function
bootstrapping
bottom half (BH) flavors
BREAKPOINT macro
breakpoints
brownouts
buffers
DMA 2nd
NIC buffer management
socket buffers
BUG() function
build scripts
building kernels
built-in kernel threads
bulk endpoints
bulk URBs
bus addresses
bus-device-driver programming interface
bus_register() function
buses
bus addresses
I²C bus transactions
LPC (Low Pin Count) bus
SMBus 2nd
SPI (Serial Peripheral Interface) bus
USB [See USB (universal serial bus).]
user space I²C/SMBus driver
w1 bus
BusyBox

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

cache
  cache misses, counting
  coherency [See DMA (Direct Memory Access).]
calibrate_delay() function
calibrating touch controllers
call_usermodehelper() function 2nd
Cambridge Silicon Radio (CSR)
CAN (controller area network)
capacity of disks, obtaining via SCSI Generic
CardBus 2nd
Card Information Structure (CIS)
cardmgr daemon
Card Services 2nd
Carrier Grade Linux (CGL)
cathode ray tube (CRT)
cdev_add() function 2nd 3rd
cdev_del() function
cdev_init() function 2nd
cdev structure 2nd
CDMA (code division multiple access) 2nd
cdrecord
cdrtools
CELF (Consumer Electronics Linux Forum)
cell phone devices
  claiming/freeing memory
  console drivers
  CPLD (Complex Programmable Logic Device)
  overview
  platform drivers
  SoC (System-on-Chip)
  USB_UART driver
  USB_UART ports
  USB_UART register layout
cellular networking
  CDMA
  GPRS
CEs (correctable errors)
CF (Compact Flash) [See also PCMCIA (Personal Computer Memory Card International Association).]
  BlueZ
  debugging
  definition
  embedded drivers
  storage
cfb_fillrect()
CFI (Common Flash Interface)
cfi_private structure
cfi_probe_chip() function
CFQ (Complete Fair Queuing) 2nd
CFS (Completely Fair Scheduler)
CGL (Carrier Grade Linux)
change markers
changing
  line disciplines
  MTU size
character drivers [See char drivers.]

char device emulation
char drivers
    accessing
    char device emulation
    CMOS driver
        I/O Control
        initialization
        internal file pointer, setting with cmos_llseek()
        opening
        overview
        reading/writing data
        register layout
        releasing
    code flow
    common problems
    data structures
    misc drivers
    overview
    parallel port communication
    parallel port LED board
        controlling with sysfs
        led.c driver
    pseudo char drivers
    RTC subsystem
    sensing data availability
        fasync() function
        overview
        select()/poll() mechanism
    sources
    UART drivers
    watchdog timer
check_bugs() function
checklist for new devices
checksums
chip drivers [See NOR chip drivers.]
Chip Select (CS)
choosing
    peripherals
    processors
Cirrus Logic EP7211 controller
CIS (Card Information Structure)
cisparse_t structure 2nd
cistpl.h file
cistpl_cftable_entry_t structure 2nd
class_create() function 2nd
class_destroy() function 2nd
class_device_add_attrs() function
class_device_create() function 2nd
class_device_create_file() function
class_device_destroy() function 2nd
class_device_register() function
class drivers
    Bluetooth
    HIDs (human interface devices)
    mass storage
    overview
    USB-Serial
classes
    device classes
    input class
    structure
clean markers
clear_bit() function
Clear To Send (CTS)

clients
    client controllers
    EEPROM device example
    PCMCIA client drivers, registering
clock_gettime() function
CLOCK_INPUT_REGISTER
clock_settime() function
close() function
CLUT (Color Look Up Table)
CLut224
CMOS_BANK0_DATA_PORT register
CMOS_BANK0_INDEX_PORT register
CMOS_BANK1_DATA_PORT register
CMOS_BANK1_INDEX_PORT register
cmos_dev structure
CMOS drivers
    I/O Control
    initialization
    internal file pointer, setting with cmos_llseek()
    opening
    overview
    reading/writing data
    register layout
    releasing
cmos_fops structure
cmos_init() function
cmos_ioctl() function
cmos_llseek() function
cmos_open() function
cmos_read() function
cmos_release() function
cmos_write() function
code division multiple access (CDMA) 2nd
code portability
codec_write_reg() function
coding styles
coldplug
collect_data() function
color modes
command-line utilities [See *specific utilities*.]
command-set 0001
command-set 0002
command-set 0020
COMMAND_REGISTER
commands [See *specific commands*.]
Common Flash Interface (CFI)
Common memory (PCMCIA)
Common UNIX Printing System (CUPS)
Compact Flash [See CF (Compact Flash).]
compact middleware
compilation
    GCC compiler
    line disciplines
complete() function 2nd
complete_all() function
complete_and_exit() function 2nd
Complete Fair Queuing (CFQ) 2nd
Completely Fair Scheduler (CFS)
completion interface
completion structure
Complex Programmable Logic Devices (CPLDs) 2nd
concurrency
    atomic operators
    CVS (Concurrent Versioning System) 2nd

debugging
NICs (network interface cards)
overview
reader-writer locks
spinlocks and mutexes
Concurrent Versioning System (CVS) 2nd
CONFIG_4KSTACKS configuration option
CONFIG_DEBUG_BUGVERBOSE configuration option
CONFIG_DEBUG_HIMEM configuration option
CONFIG_DEBUG_PAGE_ALLOC configuration option
CONFIG_DEBUG_SLAB configuration option
CONFIG_DEBUG_SPINLOCK configuration option
CONFIG_DEBUG_STACK_USAGE configuration option
CONFIG_DEBUG_STACKOVERFLOW configuration option
CONFIG_DETECT_SOFTLOCKUP configuration option
CONFIG_IKCONFIG_PROC configuration option
CONFIG_MAGIC_SYSRQ configuration option
CONFIG_MYPROJECT_FASTBOOT marker
CONFIG_MYPROJECT marker
CONFIG_PCMCIA_DEBUG() macro
config_port() function
CONFIG_PREEMPT_RT patch-set
CONFIG_PREEMPT configuration option
CONFIG_PRINTK_TIME configuration option
CONFIG_RTC_CLASS configuration option
CONFIG_SYSCTL configuration option
configuration
kernel hacking configuration options
MTD
NAND chip drivers
net_device structure
NICs
PCI resources
Wireless Extensions
configuration space (PCI), accessing
connectivity of embedded drivers
conservative governor
consistency of checksums
consistent DMA access methods
console drivers
boot logo
cell phones
consoles
Consumer Electronics Linux Forum (CELF)
container_of() function 2nd 3rd
contexts, interrupt
contrast and backlight
CONTROL_REGISTER 2nd
controller area network (CAN)
controllers
CAN (controller area network)
CS8900 controller
DRAM controllers
ECC-aware memory controller
EHCI controller
host controllers
NAND flash controllers
OTG (On-The-Go) controllers
USB device controllers
USB host controllers
coord.c application
copy_e820_map() function
copy_from_user() function 2nd
copy_to_user() function

copying system memory maps
copyleft (GNU)
correctable errors (CEs)
counters
    preemption counters
    TSC (Time Stamp Counter)
CPLDs (Complex Programmable Logic Devices) 2nd
cpqarray driver
cpufreq_register_governor() function
CPU frequency (cpufreq) driver subsystem
CPU frequency notification
cpuspeed daemon
crash command
crash dumps
create_singlethread_workqueue() function
create_workqueue() function
CRT (cathode ray tube)
crypto directory
CS (Chip Select)
CS8900 controller
cs89x0_probe1() function
cscope command
CSR (Cambridge Silicon Radio)
ctags command
CTS (Clear To Send)
CUPS (Common UNIX Printing System)
CVS (Concurrent Versioning System) 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

D-cache (Data Cache)
daemonize() function 2nd
daemons
    acpid
    cardmgr
    cpuspeed
    iscsid
    oprofiled
    pppd
    trace
DATA_REGISTER
data availability, sensing
    fasync() function
    overview
    select()/poll() mechanism
Data Cache (D-cache)
data field (sk_buff structure)
data flow, Linux-PCMCIA subsystem
data mixing (fn)
data structures [See *specific structures*.]
data transfer
    DMA data transfer
    net_device structure
    NICs (network interface cards)
    PCI
        DMA descriptors and buffers
        receiving and transmitting data
        register layout of network functions
    telemetry card example
    USB
DDWG (Digital Display Working Group)
deadline I/O scheduler 2nd
dead state (threads)
debugfs
debuggers [See kernel debuggers.]
debugging [See also ECC (error correcting code) reporting.]
    audio drivers
    block drivers
    Bluetooth
    breakpoints
    concurrency
    crash dumps
    diagnostic tools
    embedded Linux
        board rework
        debuggers
    I²C
    input drivers
    JTAG debuggers
    kdump
        example
        kexec with kdump
        setup
        sources
    kernel debuggers

downloads
entering
gdb (GNU debugger)
JTAG debuggers
kdb (kernel debugger)
kgdb (kernel GNU debugger)
overview
kernel hacking configuration options
kexec
invoking
preparation
sources
with kdump
kprobes
example
fault-handlers
inserting inside kernel functions
jprobes
kretprobes
limitations
post-handlers
pre-handlers
sources
Linux assembly
LTP (Linux Test Project)
MTD (Memory Technology Devices)
overview 2nd 3rd
PCI
PCMCIA
profiling
gprof
OProfile
overview
RAS (reliability, availability, serviceability)
test equipment
tracing
UDB (universal serial bus)
UML (User Mode Linux)
watchpoints
debug tool
DECLARE_COMPLETION() macro
DECLARE_MUTEX() function
DECLARE_WAITQUEUE() macro
DEFINE_MUTEX() function
DEFINE_TIMER() function
DEFINE_TIMER() macro
del_gendisk() function
del_timer() function
delay-loop calibration
delays
long delays
short delays
delivery
build scripts
change markers
checksum consistency
code portability
coding styles
version control
depmod utility
descriptors (USB)
detect_memory_e820() function
dev_alloc_skb() function 2nd
/dev directory

/dev names, adding to usbfs
/dev/full driver
/dev/kmem driver
/dev/mem driver
/dev/null char device
/dev/port driver
/dev/random driver
/dev/urandom driver 2nd
/dev/zero driver
dev_kfree_skb() function
dev_t structure
devfs
device checklist
device classes
device controllers
device_driver structure
device_register() function
devices [See also *specific devices*.]
ACPI (Advanced Configuration and Power Interface) devices
interrupt handling [See interrupt handling.]
Linux device model
device classes
hotplug/coldplug
kobjects
microcode download
module autoload
overview
sysfs
udev 2nd
memory barriers
power management
diagnostic tools
dialup networking (DUN)
die_chain structure
die notifications 2nd
diff command
Digital Display Working Group (DDWG)
Digital Visual Interface (DVI)
direct-to-home (DTH) interface
Direct Memory Access [See DMA (Direct Memory Access).]
directories [See also *specific directories*.]
disable_irq() function 2nd
disable_irq_nosync() function 2nd
disabling IRQs (interrupt requests)
disconnecting telemetry drivers
Disk-On-Modules (DOMs)
disk capacity, obtaining via SCSI Generic
disk mirroring
display architecture
displaying images with mmap()
display parameters
distributions
dma_addr_t structure
DMA_ADDRESS_REGISTER
DMA (Direct Memory Access) [See also Ethernet-Modem card example.]
buffers
consistent DMA access methods
definition
descriptors and buffers
IOMMU (I/O memory management unit)
masters
navigation systems
scatter-gather
streaming DMA access methods

synchronous versus asynchronous
dma_map_single() function
DMA_RX_REGISTER
dma_set_mask() function
DMA_SIZE_REGISTER
DMA_TX_REGISTER
DMA data transfer
dmix (fn)
do_gettimeofday() function 2nd
do_ida_intr() function
do_IRQ() function
do_map_probe() function
documentation
Documentation directory
procfs
seq files
dollar sign ($)
domain-specific electronics
DOMs (Disk-On-Modules)
dongles, Infrared
doorbells
DOS debug tool
down() function
down_read() function
down_write() function
DRAM controllers
DRDs (dual-role devices)
driver_register() function
drivers directory
Driver Services
ds (driver services) module
DTH (direct-to-home) interface
dual-role devices (DRDs)
dump_port() function
DUN (dialup networking)
dv1394 driver
DVI (Digital Visual Interface)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

e820.c file
e820.h file
e1000 PCI-X Gigabit Ethernet driver
ECC (error correcting code) reporting 2nd
  correctable errors (CEs)
  ECC-aware memory controller
  ECC-related registers on DRAM controller
  edac_mc module
  embedded drivers
  multibit errors (MBEs)
  single-bit errors (SBEs)
  /sys/devices/system/edac/ directory
  uncorrectable errors (UEs)
ECs (embedded controllers)
EDAC (Error Detection and Correction) 2nd
  correctable errors (CEs)
  ECC-aware memory controller
  ECC-related registers on DRAM controller
  edac_mc module
  embedded drivers
  error-handling aids
  multibit errors (MBEs)
  single-bit errors (SBEs)
  /sys/devices/system/edac/ directory
  uncorrectable errors (UEs)
edac_mc module
edge-sensitive devices
eep_attach() function
eep_probe() function
eep_read() function
EEPROM device example
  accessing
  adapter capabilities, checking
  clients, attaching
  i2c_del_driver() function
  initializing
  ioctl() function
  llseek() method
  memory banks
  opening
  overview
  probing
  RFID (Radio Frequency Identification) transmitters
EHCI (Enhanced Host Controller Interface) 2nd
EISA (Extended Industry Standard Architecture)
elv_next_request() function 2nd
embedded bootloaders
  BLOB (BootLoader Object)
  bootstrapping
  GRUB
  LILO (Linux Loader)
  overview
  RedBoot
  SYSLINUX
  table of

embedded controllers (ECs)
embedded drivers
  audio
  brownouts
  buttons and wheels
  connectivity
  CPLDs (Complex Programmable Logic Devices)
  domain-specific electronics
  ECC capabilities
  flash memory
  FPGAs (Field Programmable Gate Arrays)
  overview
  PCMCIA/CF 2nd
  PWM (pulse-width modulator) units
  RTC
  SD/MMC
  touch screens
  UARTs
  udev
  USB
  video
embedded Linux
  challenges
  component selection
  debugging
    board rework
    debuggers
  embedded bootloaders
    BLOB (BootLoader Object)
    bootstrapping
    GRUB
    LILO (Linux Loader)
    overview
    RedBoot
    SYSLINUX
    table of
  embedded drivers
    audio
    brownouts
    buttons and wheels
    connectivity
    CPLDs (Complex Programmable Logic Devices)
    domain-specific electronics
    ECC capabilities
    flash memory
    FPGAs (Field Programmable Gate Arrays)
    overview
    PCMCIA/CF
    PWM (pulse-width modulator) units
    RTC
    SD/MMC
    touch screens
    UARTs
    USB
    video
  hardware block diagram
  kernel porting
  memory layout
  overview
  root filesystem
    compact middleware
    NFS-mounted root
    overview
  test infrastructure

tool chains
USB (universal serial bus)
emulation
block device emulation
char device emulation
enable_irq() function 2nd
enabling IRQs (interrupt requests)
end field (sk_buff structure)
end_request() function
endpoint addresses
endpoints (USB)
Enhanced Host Controller Interface (EHCI) 2nd
enumeration
EP7211 controller
epoll() function
erase_info_user structure
erase_info structure
error correcting codes (ECCs) [See ECC (error correcting code) reporting.]
Error Detection And Correction [See EDAC (Error Detection and Correction).]
/etc/inittab file
/etc/rc.sysinit
etags command
eth1394 driver
Ethernet-Modem card example
data transfer
DMA descriptors and buffers
receiving and transmitting data
register layout of network functions
modem functions
probing
registering
MODULE_DEVICE_TABLE() macro
network functions
probing
registering
PCI_DEVICE() macro
pci_device_id structures
Ethernet NIC driver
ethtool
ethtool_ops structure 2nd
evbug module
Evdev interface
events
input event drivers
Evdev interface
overview
virtual mouse device example
writing
LTT events
notifier event handlers
events/n threads
evolution of Linux
eXecute In Place (XIP)
EXIT_DEAD state
EXIT_ZOMBIE state
expired queues
ExpressCards 2nd
EXT3 filesystem
EXT4 filesystem
eXtended Graphics Array (XGA)
Extended Industry Standard Architecture (EISA)
external watchdogs

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

fasync() function
fasync_helper() function 2nd
fault-handlers (kprobes)
fb_blank() method
fb_check_var() method
fb_fillrect()
fb_var_screeninfo structure
FCC (Federal Communications Commission)
fcntl() function
Federal Communications Commission (FCC)
Fibre Channel
Field Programmable Gate Arrays (FPGAs)
FIFO (first-in first-out) memory
file_operations structure 2nd 3rd
file structure
filesystems
    debugfs
    EXT3
    EXT4
    JFFS (Journaling Flash File System)
    NFS (Network File System)
    procfs [See procfs.]
    rootfs
        compact middleware
        NFS-mounted root
        obtaining
        overview
    sysfs
    usbfs virtual filesystem
    VFS (Virtual File System) 2nd
    YAFFS (Yet Another Flash File System)
File Translation Layer (FTL)
Finite State Machine (FSM)
FireWire
Firmware Hub (FWH)
first-in first-out (FIFO) memory
flash_eraseall command
flash memory [See also MTD (Memory Technology Devices).]
    CFI-compliant flash, querying
    definition
    embedded drivers
    NAND
    NOR
    sectors
floppy storage
flow control (NICs)
flush_buffer() function
flushing data
forums
FPGAs (Field Programmable Gate Arrays)
frame buffer API
frame buffer drivers
    accelerated methods
    color modes
    contrast and backlight

    data structures
    DMA
    parameters
    screen blanking
free_irq() function 2nd
free_netdev() function
freeing
    IRQs (interrupt requests)
    memory
Freescale MC13783 Power Management and Audio Component (PMAC)
Freescale MPC8540
Free Software Foundation
frequency scaling
Front Side Bus (FSB)
fs directory
FSM (Finite State Machine)
fsync() function
FTDI driver
FTL (File Translation Layer)
full char device
full-speed USB
function controllers
functions [See *specific functions*.]
FWH (Firmware Hub)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

gadget drivers
garbage collector (GC)
GAS (GNU Assembler)
GC (garbage collector)
GCC compiler
GCC Inline Assembly HOWTO
gdb (GNU debugger)
gendisk structure 2nd
general-purpose mouse (gpm)
General Object Exchange Profile (GOEP)
General Packet Radio Service (GPRS) 2nd
General Purpose I/O (GPIO)
generating
    patches
    preprocessed source code
GET_DEVICE_ID command
get_random_bytes() function
get_stats() method
get_wireless_stats() function
getitimer() function
gettimeofday() function
Glibc libraries
Global System for Mobile Communication (GSM) 2nd
glow_show_led() function 2nd
GMCH (Graphics and Memory Controller Hub)
GNU
    copyleft
    GAS (GNU Assembler)
    gdb (GNU debugger)
    LGPL (Lesser General Public License)
    GPL (GNU Public License)
GOEP (General Object Exchange Profile)
governors
GPIO (General Purpose I/O)
GPL (GNU Public License)
gpm (general-purpose mouse)
gprof
GPRS (General Packet Radio Service) 2nd
Graphics and Memory Controller Hub (GMCH)
GRUB
GSM (Global System for Mobile Communication) 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

HA (High Availability) project
HAL (Hardware Access Layer)
halt (HLT) instruction
ham radio
handle_IRQ_event() function
handling interrupts [See interrupt handling.]
hard-specific modules (HDMs)
hard_start_xmit() function
hard_start_xmit method
Hard Drive Active Protection System (HDAPS)
Hardware Access Layer (HAL)
hardware block diagrams
  embedded system
  PC-compatible system
hardware RAID
hash lists
HCI (Host Control Interface) 2nd
hci_uart.ko
HD (High Definition) Audio
HDAPS (Hard Drive Active Protection System)
HDLC (High-level Data Link Control)
HDMI (High-Definition Multimedia Interface)
HDMs (hard-specific modules)
hdparm utility
HDTV (High-Definition Television)
head field (sk_buff structure)
helper interfaces
  completion interface
  error-handling aids
  hash lists
  kthread helpers
  linked lists
    creating
    data structures, initializing
    functions
    work submission
    worker thread
  notifier chains
  overview
  work queues
hidp driver
HIDs (human interface devices) 2nd 3rd 4th
High-Definition Multimedia Interface (HDMI)
High-Definition Television (HDTV)
High-level Data Link Control (HDLC)
high-speed interconnects
  InfiniBand
  RapidIO
    Fibre Channel
    iSCSI (Internet SCSI)
  USB
High Availability (HA) project
High Definition (HD) Audio
high memory
history of Linux

hlist_head structure 2nd
hlist_nodes structure
hlists (hash lists)
HLT instruction
HNP (Host Negotiation Protocol)
host adapters
Host Control Interface (HCI) 2nd
Host Negotiation Protocol (HNP)
hotplug
hubs, root
human interface devices (HIDs) 2nd 3rd 4th
hwclock command
HZ 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

I-cache (Instruction Cache)
I/O Control
    CMOS driver
    touch controller
I/O memory management unit (IOMMU)
I/O regions
    accessing
    dumping bytes from
I/O schedulers 2nd
I²C [See also SMBus.]
    1-wire protocol
    bus transactions
    compared to USB
    core
    debugging
    definition
    EEPROM device example
        accessing
        adapter capabilities, checking
        clients, attaching
        i2c_del_driver() function
        initializing
        ioctl() function
        llseek() method
        memory banks
        opening
        overview
        probing
        RFID (Radio Frequency Identification) transmitters
    i2c-dev
    LM-Sensors
    overview
    RTC (Real Time Clock)
    sources
    SPI (Serial Peripheral Interface) bus
    summary of data structures
    summary of kernel programming interfaces
    user mode I²C
i2c-dev module 2nd
i2c_add_adapter() function
i2c_add_driver() function 2nd
i2c_attach_client() function
i2c_check_functionality() function 2nd
i2c_client_address_data structure 2nd
i2c_client structure
i2c_del_adapter() function
i2c_del_driver() function 2nd
i2c_detach_client() function
i2c_driver structure
i2c_get_functionality() function 2nd
i2c_msg structure
i2c_probe() function
i2c_smbus_read_block_data() function
i2c_smbus_read_byte() function
i2c_smbus_read_byte_data() function 2nd

i2c_smbus_read_word_data() function
i2c_smbus_write_block_data() function
i2c_smbus_write_byte() function
i2c_smbus_write_byte_data() function
i2c_smbus_write_quick() function
i2c_smbus_write_word_data() function 2nd
i2c_transfer() function 2nd
I2O (Intelligent Input/Output)
I2O SIG (I2O Special Interest Group)
I2S (Inter-IC Sound) bus
i386 boot assembly code
I855_EAP_REGISTER register
I855_ERRSTS_REGISTER register
IDE (Integrated Drive Electronics)
IEEE 1394
images
  displaying with mmap()
  initramfs
imx.c driver
in[b|w|l|sn|sl]() function
in_interrupt() function
inb() function 2nd 3rd
include/asm-x86/e820.h file
include/pcmcia/cistpl.h file
include directory
Industries Standard Architecture (ISA)
InfiniBand
Infrared 2nd 3rd
  data structures
  dongles
  IrCOMM
  IrDA sockets
  kernel programming interfaces
  Linux-IrDA
  LIRC
  networking
  sources
  Super I/O chip
infrastructure mode (WLAN)
init() function
  char drivers
  CMOS driver
  EEPROM device example
init_completion() function 2nd
init directory
INIT_LIST_HEAD() function
init_MUTEX() function
init_timer() function 2nd
initialization
  CMOS driver
  EEPROM device example
  myblkdev storage controller
  telemetry configuration register
  telemetry driver
initiators (iSCSI)
init process
initramfs root filesystem
initrd memory
inittab file
inl() function 2nd
inline assembly
input_allocate_device() function
input_dev structure
input_event() function

input_event structure
input_handler structure 2nd
input_register_device() function 2nd 3rd 4th
input_register_handler() function
input_report_abs() function 2nd
input_report_key() function
input_report_rel() function
input_sync() function 2nd
input_unregister_device() function
input class
input drivers
    debugging
    input device drivers
        accelerometers
        Bluetooth keyboards
        Bluetooth mice
        output events
        PC keyboards
        PS/2 mouse
        roller mouse device example
        serio
        touch controllers
        touchpads
        trackpoints
        USB keyboards
        USB mice
    input event drivers
        Evdev interface
        overview
        virtual mouse device example
        writing
    input subsystem
    sources
    summary of data structures
input subsystem
insmod command
Instruction Cache (I-cache)
int 0x15 service 2nd
Integrated Drive Electronics (IDE)
Intelligent Input/Output (I2O)
Inter-IC Sound (I2S) bus
Inter-Integrated Circuit [See I^2C.]
internal file pointer, setting with cmos_llseek()
Internet address notification
Internet Protocol (IP)
Internet SCSI (iSCSI)
interrupt contexts 2nd
interrupt handling
    asynchronous interrupts
    block drivers
    interrupt contexts
    IRQs (interrupt requests)
        assigning
        definition
        enabling/disabling
        freeing
        requesting
    overview
    roller wheel device example
        edge sensitivity
        free_irq() function
        request_irq() function
        roller interrupt handler
        softirqs

tasklets
wave forms generated by
softirqs
synchronous interrupts
tasklets
interruptible state (threads)
interrupt requests [See IRQs (interrupt requests).]
interrupts
interrupt service routine (ISR)
invoking kexec
inw() function 2nd
ioctl() function 2nd 3rd 4th 5th
IOMMU (I/O memory management unit)
ioperm() function 2nd
iopl() function 2nd
ioremap() function
ioremap_nocache() function 2nd
iovec structure
IP (Internet Protocol)
ipc directory
ipx_routes_lock
IrCOMM
irda-utils package
IrDA socket (IrSock) 2nd
IrLAP (IR Link Access Protocol)
IrLMP (IR Link Management Protocol)
irq command
IRQ_HANDLED flag
IRQF_DISABLED flag
IRQF_SAMPLE_RANDOM flag
IRQF_SHARED flag
IRQF_TRIGGER_HIGH flag
IRQF_TRIGGER_RISING flag
IRQs (interrupt requests)
assigning
cell phone device example
definition
enabling/disabling
freeing
requesting
roller wheel device example
IrSock (IrDA socket)
IS_ERR() function 2nd
ISA (Industries Standard Architecture)
ISA NICs
iSCSI (Internet SCSI)
iscsi_tcp.c driver
iscsid daemon
ISR (interrupt service routine)
iterator methods
next()
show()
start()
stop()

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

JFFS (Journaling Flash File System)
jiffies 2nd
Journaling Flash File System (JFFS)
jprintk() function
jprobe_return() function
jprobes
JTAG (Joint Test Action Group)
    debuggers 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

kacpid thread
kallsyms_lookup_name() function
kapmd thread
kbnepd
kdb (kernel debugger)
kdump
  example
  kexec with kdump
  setup
  sources
kernel.org
kernel_thread() function 2nd
kernel debuggers
  downloads
  entering
  gdb (GNU debugger)
  JTAG debuggers
  kdb (kernel debugger)
  kgdb (kernel GNU debugger)
  overview
kernel directory
kernel hacking configuration options
kernel mode
kernel modules [See modules.]
kernel probes [See kprobes.]
kernel processes [See kernel threads.]
kernel programming interfaces [See *specific functions*.]
kernels
  boot process
    BIOS-provided physical RAM map
    delay-loop calibration
    EXT3 filesystem
    HLT instruction
    I/O scheduler
    init process
    initrd memory
    kernel command line
    Linux boot sequence
    low memory/high memory
    PCI resource configuration
    registered protocol families
    start_kernel() function
  building
  concurrency
    atomic operators
    debugging
    overview
    reader-writer locks
    spinlocks and mutexes
  data structures, table of
  debuggers
    downloads
    entering
    gdb (GNU debugger)
    JTAG debuggers

kdb (kernel debugger)
kgdb (kernel GNU debugger)
overview
helper interfaces
completion interface
error-handling aids
hash lists
kthread helpers
linked lists
notifier chains
overview
work queues
interrupt contexts
kernel.org repository
kernel hacking configuration options
kernel mode
kernel programming interfaces, table of
memory allocation
modules
edac_mc
loading
porting
process contexts
sources 2nd
source tree layout
directories 2nd
navigating
threads
bdflush
creating
definition
events/n threads
kacpid
kapmd
kjournald
ksoftirqd/0
kthreadd
kthread helpers
kupdated
listing active threads
nfsd
pccardd
pdflush
process states
user mode helpers
wait queues
timers
HZ
jiffies
long delays
overview
RTC (Real Time Clock)
short delays
TSC (Time Stamp Counter)
uClinux
user mode
kernel threads
bdflush
creating
definition
events/n threads
kacpid
kapmd
kjournald

- ksoftirqd/0
- kthreadd
- kthread helpers
- kupdated
- listing active threads
- nfsd
- pccardd
- pdflush
- process states
- user mode helpers
- wait queues

kernel timers
- HZ
- jiffies
- long delays
- overview
- RTC (Real Time Clock)
- short delays
- TSC (Time Stamp Counter)

kerneltrap.org

kexec
- invoking
- preparation
- sources
- with kdump

kexec-tools package

keyboards
- Bluetooth keyboards
- overview
- PC keyboards
- USB keyboards

keycodes

keypads

kfree() function

kgdb (kernel GNU debugger)

kgdbwait command

khubd

kill_fasync() function 2nd

kjournald thread

kmalloc() function 2nd 3rd 4th

kmem char device

kobj_type structure 2nd

kobject_add() function

kobject_register() function 2nd

kobject_uevent() function

kobject_unregister() function 2nd

kobjects 2nd

kprobes
- example
  - *kprobe handlers, registering*
  - *mydrv.c file*
  - *patches, inserting*
- fault-handlers
- inserting inside kernel functions
- jprobes
- kretprobes
- limitations
- post-handlers
- pre-handlers
- sources

kref_get() function

kref_init() function

kref_put() function

kref object

kret_tty_open() function
kretprobes
kset structure
ksoftirqd/0 kernel thread
kthread_create() function 2nd
kthread_run() function
kthread_should_stop() function 2nd
kthread_stop() function
kthreadd kernel thread
kthread helpers
ktype_led structure
kupdated kernel thread
kzalloc() function 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

L2CAP (Logical Link Control and Adaptation Protocol)
l2cap.ko
LAD (Linux Audio Developers) list
LAN Emulation (LANE)
LANs (local area networks) 2nd
laptops
large procfs reads
layered architecture (serial drivers)
LBA (logical block addressing)
LCDC (Liquid Crystal Display Controller)
LCD controllers
ldisc.read() function
ldisc.receive_buf() function
led.c driver
led_init() function
led_write() function
LED board [See parallel port LED boards.]
legacy drivers
    BIOS
    RTC driver
len field (sk_buff structure)
level-sensitive devices
LGPL (GNU Lesser General Public License)
libATA
lib directory
libraries
    alsa-lib
    Glibc
    libraw1394
libraw1394 library
libusb programming template
likely() function 2nd
LILO (Linux Loader)
line disciplines (touch controller device example)
    changing
    compiling
    connection diagram
    flushing data
    I/O Control
    open/close operations
    opening
    overview
    read paths
    unregistering
    write paths
linked lists
    creating
    data structures, initializing
    functions
    worker thread
    work submission
links (PCIe)
linux.conf.au
Linux Amateur Radio AX.25 HOWTO
Linux assembly

boot sequence
debugging
GNU Assembler (GAS)
i386 boot assembly code
inline assembly
Microsoft Macro Assembler (MASM)
Netwide Assembler (NASM)
Linux Asynchronous I/O (AIO)
linux-audio-dev mailing list
Linux Audio Developers (LAD) list
Linux device model
device classes
hotplug/coldplug
kobjects
microcode download
module autoload
overview
sysfs
udev 2nd
Linux distributions
Linux history and development
linux-ide mailing list
Linux-IrDA
Linux Kernel Crash Dump (LKCD)
Linux Kernel Mailing List (LKML)
Linux Kongress
Linux Loader (LILO)
Linux-MTD JFFS HOWTO
linux-mtd mailing list
Linux-MTD subsystem [See MTD (Memory Technology Devices).]
Linux-PCMCIA subsystem [See PCMCIA (Personal Computer Memory Card International Association).]
linux-scsi mailing list
Linux Symposium
Linux Test Project (LTP) 2nd
Linux Trace Toolkit [See LTT (Linux Trace Toolkit).]
Linux Trace Toolkit Viewer (LTTV)
linux-usb-devel mailing list 2nd
Linux-USB subsystem [See USB (universal serial bus).]
Linux-video subsystem
LinuxWorld Conference and Expo
Liquid Crystal Display Controller (LCDC)
LIRC (Linux Infrared Remote Control)
list_add() function
list_add_tail() function
list_del() function 2nd
list_empty() function
list_entry() function 2nd
list_for_each_entry() function 2nd
list_for_each_entry_safe() function 2nd
list_head structure 2nd
list_replace() function
list_splice() function
lists
hash lists
linked lists
creating
data structures, initializing
functions
worker thread
work submission
LKCD (Linux Kernel Crash Dump)
LKML (Linux Kernel Mailing List)
llseek() function 2nd
LM-Sensors

loading modules
loadkeys
local_irq_disable() function
local_irq_enable() function 2nd
local_irq_restore() function
local_irq_save() function
local area networks (LANs) 2nd
localtime() function
locks
lockups, soft
log command
logical addresses
logical block addressing (LBA)
Logical Link Control and Adaptation Protocol (L2CAP)
long delays
loopback devices
loops_per_jiffy variable 2nd 3rd
low-speed USB
low-voltage differential signaling (LVDS)
low memory
Low Pin Count (LPC) bus
lp.c driver
lp_write() function
LPC (Low Pin Count) bus
lseek() function
lsmod command
lspci command
lsvpd utility
LTP (Linux Test Project) 2nd
LTT (Linux Trace Toolkit)
    components
    events
    LTTng
    LTTV (Linux Trace Toolkit Viewer)
    trace dumps
LTTng
LTTV (Linux Trace Toolkit Viewer)
LVDS (low-voltage differential signaling)
lwn.net
lxr command

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

MAC (Media Access Control) addresses
macros [See *specific macros*.]
Madplay 2nd
mailboxes (RapidIO)
mailing lists 2nd
maintenance
  build scripts
  change markers
  checksum consistency
  code portability
  coding styles
  version control
major numbers (char drivers)
make command
MAN (metropolitan area network)
map_info structure 2nd
map drivers
  definition
  MTD partition maps, creating
  probe method
  registering
mapping memory
maps, system memory map
  copying
  obtaining
markers, clean
MASM (Microsoft Macro Assembler)
mass storage devices (USB)
Master Boot Record (MBR)
Master In Slave Out (MISO)
Master Out Slave In (MOSI)
masters (DMA)
maximum transmission unit (MTU) 2nd
mb() function
MBEs (multibit errors)
MBR (Master Boot Record)
MCA (Micro-Channel Architecture)
MCH (Memory Controller Hub)
md command
mdelay() function
media_changed() method
Media Access Control (MAC) addresses
Media Independent Interface (MII)
mem char device
MEMERASE command
MEMLOCK command
memory
  accessing from user space
  allocating
  cache misses, counting
  claiming/freeing
  CMOS (complementary metal oxide semiconductor)
  DMA (Direct Memory Access) [See also Ethernet-Modem card example.]
    buffers
    consistent DMA access methods

definition
IOMMU (I/O memory management unit)
masters
scatter-gather
streaming DMA access methods
synchronous versus asynchronous
embedded Linux memory layout
FIFO (first-in first-out) memory
flash memory
CFI-compliant flash, querying
definition
embedded drivers
NAND
NOR
sectors
high memory
initrd memory
low memory
mapping
memory barriers
memory zones
MTD (Memory Technology Devices)
flash memory
illustration of Linux-MTD subsystem
map drivers
MTD core
NAND drivers
NOR Chip drivers
overview
partition maps, creating
User Modules
pages
system memory map
copying
obtaining
zero page
ZONE_DMA
ZONE_HIGH
ZONE_NORMAL
memory.c file
memory banks (EEPROM)
Memory Controller Hub (MCH)
memory_cs Card Services driver
Memory Technology Devices [See MTD (Memory Technology Devices).]
memory zones
MEMUNLOCK command
memwalkd() function
methods [See *specific methods*.]
metropolitan area network (MAN)
mice
Bluetooth mice
PS/2 mouse
roller mouse device example
touchpads
trackpoints
USB mice
virtual mouse device example
gpm (general-purpose mouse)
vms.c input driver
Micro-Channel Architecture (MCA)
microcode download
microdrives
Microsoft Macro Assembler (MASM)
middleware

MII (Media Independent Interface)
million instructions per second (MIPS)
MIMO (Multiple In Multiple Out)
minicom
Mini PCI
minor numbers (char drivers)
MIPS (million instructions per second)
mirroring disks
misc_deregister() function
misc_register() function 2nd 3rd
Miscdevice structure
misc (miscellaneous) drivers [See also watchdog timer.]
MISO (Master In Slave Out)
mixers
mkinitramfs command
mkinitrd command
mktime() function
mlockall() function 2nd
-mm patch
mmap() function 2nd 3rd
mmapping
MMC (MultiMediaCard)
mm directory
mod_timer() function 2nd
modem functions
  probing
  registering
modes
  kernel mode
  protected mode
  real mode
  user mode
modinfo command
modprobe command
MODULE_DEVICE_TABLE() macro 2nd 3rd
modules
  autoloading
  edac_mc
  loading
Molnar, Ingo
Morton, Andrew
MOSI (Master In Slave Out)
most significant bit (MSB)
mouse_poll() function
mousedev
Moving Picture Experts Group (MPEG) 2nd
MP3 player example
  ALSA driver code listing
  ALSA programming
  codec_write_reg() function
  MP3 decoding complexity
  mycard_audio_probe() function
  mycard_audio_remove() functions
  mycard_hw_params() function
  mycard_pb_prepare() function
  mycard_pb_trigger() function
  mycard_playback_open() function
  overview
  register layout of audio hardware
  snd_card_free() function
  snd_card_new() function
  snd_card_proc_new() function
  snd_card_register() function
  snd_ctl_add() function

snd_ctl_new1() function
snd_device_new() function
snd_kcontrol structure
snd_pcm_hardware structure
snd_pcm_lib_malloc_pages() function
snd_pcm_lib_preallocate_pages_for_all() function
snd_pcm_new() function
snd_pcm_ops structure
snd_pcm_set_ops() function
user programs
MPC8540 (Freescale)
MPEG (Moving Picture Experts Group) 2nd
MPLS (Multiprotocol Label Switching)
MPoA (Multi Protocol over ATM)
MSB (most significant bit)
msleep() function
msync() function
MTD (Memory Technology Devices)
configuration
data structures
debugging
flash memory
FWH (Firmware Hub)
illustration of Linux-MTD subsystem
kernel programming interfaces
map drivers
definition
MTD partition maps, creating
overview
probe method
registering
MTD core
NAND chip drivers
block size
configuring
definition
layout
NAND flash controllers
OOB (out-of-band) information
page size
spare area
NOR chip drivers
definition
querying CFI-compliant flash
partition maps, creating
sources
User Modules
block device emulation
char device emulation
definition
JFFS (Journaling Flash File System)
MTD-utils
overview
YAFFS (Yet Another Flash File System)
XIP (eXecute In Place)
mtd_info structure
mtd_partition structure 2nd
MTD-utils
mtdblock driver
mtdchar driver
MTU (maximum transmission unit) 2nd
multibit errors (MBEs)
MultiMediaCard (MMC)
multimeters

Multiple In Multiple Out (MIMO)
Multiprotocol Label Switching (MPLS)
Multi Protocol over ATM (MPoA)
munmap() function
mutex_init() function
mutex_lock() function
mutex_unlock() function
mutexes 2nd
mutual exclusion (mutexes)
my_dev_event_handler() function
my_device_xmit() function
my_die_event_handler() function
my_noti_chain structure
my_release() function
myblkdev_init() function
myblkdev_ioctl() function
myblkdev_request() function
myblkdev storage controller
    block device operations
    disk access
    initialization
    overview
    register layout
mycard_audio_probe() function
mycard_audio_remove() function
mycard_change_mtu() function
mycard_get_eeprom() function
mycard_get_stats() function
mycard_hw_params() function
mycard_pb_prepare() function
mycard_pb_trigger() function
mycard_pb_vol_info() function
mycard_playback_open() function
mydrv.c file
mydrv_dev structure
mydrv_init() function
mydrv_worker() function
mydrv_workitem structure
mydrv_wq structure
myevent_id structure
myevent_waitqueue structure
myrtc_attach() function
myrtc_gettime() function

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

N_TCH line discipline 2nd
n_touch_chars_in_buffer() function
n_touch_open() function
n_touch_receive_buf() function
n_touch_receive_room() function
n_touch_write() function
n_touch_write_wakeup() function
nand_ecclayout structure 2nd
nand_flash_ids[] table
NAND chip drivers
  block size
  configuring
  definition
  layout
  NAND flash controllers
  OOB (out-of-band) information
  page size
  spare area
NAND File Translation Layer (NFTL)
NAND flash controllers
NAND flash memory
NAND storage
nanosleep() function
NAPI (New API) 2nd
NASM (Netwide Assembler)
navigation
  frame buffer drivers
    accelerated methods
    color modes
    contrast and backlight
    data structures
    DMA
    parameters
    screen blanking
  source tree layout
NCP (Network Control Protocol)
ndelay() function
net_device_stats structure 2nd
net_device method
net device notification
net_device structure
  activation
  bus-specific methods
  configuration
  data transfer
  overview
  statistics
  watchdog timeout
net directory
netdev_chain structure
netif_device_attach() function
netif_device_detach() function
netif_queue_stopped() function 2nd
netif_receive_skb() function
netif_rx() function 2nd 3rd

netif_rx_complete() function 2nd
netif_rx_schedule() function
netif_rx_schedule_prep() function
netif_start_queue() function 2nd
netif_stop_queue() function 2nd
netif_wake_queue() function 2nd
Netlink sockets
netperf
Netrom
Netwide Assembler (NASM)
Network Control Protocol (NCP)
Network File System (NFS)
network interface cards [See NICs (network interface cards).]
networks
  Bluetooth 2nd 3rd
  Infrared
  LANs (local area networks)
  network functions
    probing
    registering
  NICs (network interface cards) [See NICs (network interface cards).]
  throughput
    driver performance
    overview
    protocol performance
New API (NAPI)
new device checklist
next() function
NFS (Network File System) 2nd
nfs_unlock_request() function
nfsd kernel thread
NFTL (NAND File Translation Layer)
nice values
NICs (network interface cards)
  activation
  ATM (asynchronous transfer mode)
  buffer management
  concurrency control
  configuration
  data structures
  data transfer
  Ethernet NIC driver
  ISA NICs
  MTU size, changing
  net device interface [See net_device structure.]
  network throughput
    driver performance
    overview
    protocol performance
  overview
  protocol layers
    flow control
    receive path
    transmit path
  socket buffers
  sources
  statistics
  summary of kernel programming interfaces
  watchdog timeout
Noop 2nd
NOR chip drivers
  definition
  querying CFI-compliant flash
NOR flash memory

North Bridge
notebooks
notifications
    CPU frequency notification
    die notification
    Internet address notification
    net device notification
    notifier chains
notifier_block structure
notifier chains
null sink
NVRAM drivers, updating with seq files

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

O(1) scheduler
OBEX (OBject EXchange)
objdump command
OBject EXchange (OBEX)
objects, kobjects
obtaining system memory map
OEMs (original equipment manufacturers)
off-the-shelf (OTS) modules
OHCI (Open Host Controller Interface)
ohci1394 driver
On-The-Go (OTG) controllers
ondemand governor
OOB (out-of-band) information
opcontrol
open() method
  block drivers
  CMOS driver
  EEPROM driver
  net_device structure
open_softirq() function
Open Host Controller Interface (OHCI)
opening
  CMOS driver
  EEPROM driver
  touch controllers
Open Sound System (OSS)
Open Source Development Lab (OSDL)
Open Systems Interconnect (OSI)
operators, atomic
opreport
OProfile 2nd
  cache misses, counting
  opcontrol
  opreport
oprofiled daemon
original equipment manufacturers (OEMs)
OS-specific modules (OSMs)
oscilloscopes
OSDL (Open Source Development Lab)
OSI (Open System Connect)
OSMs (OS-specific modules)
OSS (Open Sound System)
OTG (On-The-Go) controllers
out-of-band (OOB) information
outb() function 2nd 3rd
outl() function 2nd
outsl() function
outsn() function
output events (input device drivers)
outw() function 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

packages
    alsa-utils
    kexec-tools
    MTD-utils
    pcmcia-cs
    pcmciautils
    sysfsutils
pages (memory)
PAN (personal area network)
Parallel ATA (PATA)
parallel port communication
parallel port LED boards
    controlling from user space
    controlling with sysfs
    led.c driver
parallel printer drivers
Pardevice structure
parport
parport_claim_or_block() function
parport_read_data() function
parport_register_device() function 2nd
parport_register_driver() function 2nd
parport_release() function
parport_unregister_device() function
parport_unregister_driver() function
parport_write_data() function
partitions
    MTD partition maps, creating
    swap space
PATA (Parallel ATA)
patches
    applying
    CONFIG_PREEMPT_RT patch-set
    creating
    definition
    kernel.org repository
patch utility
PC-compatible system hardware block diagram
PCBs (printed circuit boards)
pccardctl command
pccardd thread
PC Cards
PC keyboards
PCI (Peripheral Component Interconnect)
    accessing PCI regions
        configuration space
        I/O and memory regions
    addressing and identification
    CardBus 2nd
    compared to USB
    data structures
    debugging
    definition
    DMA (Direct Memory Access)
        buffers

consistent DMA access methods
definition
descriptors and buffers
IOMMU (I/O memory management unit)
masters
scatter-gather
streaming DMA access methods
synchronous versus asynchronous
Ethernet-Modem card example
data transfer
modem functions, probing
modem functions, registering
MODULE_DEVICE_TABLE() macro
network functions, probing
network functions, registering
PCI_DEVICE() macro
pci_device_id structures
Express Cards
kernel programming interfaces
Mini PCI
PCI-based solutions
PCI Express
PCI Express Mini Card
PCI Extended (PCI-X)
PCI inside South Bridge system
resources, configuring
serial communication
sources
pci_alloc_consistent() function 2nd
PCI_DEVICE() macro 2nd
pci_device_id structure 2nd 3rd
pci_dev structure 2nd
pci_disable_device() function
pci_dma_sync_sg() function
pci_dma_sync_single() function
pci_driver structure
pci_enable_device() function
PCI Express 2nd
PCI Express Mini Card
PCI Extended (PCI-X)
pci_free_consistent() function
pci_iomap() function 2nd
pci_map_page() function
pci_map_sg() function 2nd
pci_map_single() function 2nd
pci_read_config_byte() function 2nd
pci_read_config_dword() function 2nd
pci_read_config_word() function 2nd
pci_register_driver() function 2nd
pci_request_region() function 2nd
pci_resource_end() function 2nd
pci_resource_flags() function 2nd
pci_resource_len() function 2nd
pci_resource_start() function 2nd
pci_set_dma_mask() function
pci_set_master() function
pci_unmap_sg() function
pci_unmap_single() function
pci_unregister_driver() function
pci_write_config_byte() function 2nd
pci_write_config_dword() function 2nd
pci_write_config_word() function 2nd
PCI-X (PCI Extended)
PCIe (PCI Express)

PCM (pulse code modulation)
PCMCIA (Personal Computer Memory Card International Association)
  Attribute memory
  CardBus devices
  Card Services
  CIS (Card Information Structure)
  client drivers, registering
  Common memory
  data-flow path between components
  data structures
    cisparse_t
    cistpl_cftable_entry_t
    pcmcia_device
    pcmcia_device_id
    pcmcia_driver structure
    summary of
    tuple_t
  debugging
  definition
  device IDs and hotplug methods
  Driver Services
  driver services module (ds)
  embedded drivers
  ExpressCards
  kernel programming interfaces
  Linux-PCMCIA subsystem interaction
  mailing list
  on embedded systems
  on laptops
  pcmciautils package
  serial PCMCIA
  sources
  storage
  udev
pcmcia-cs package
pcmcia_device_id structure 2nd
PCMCIA_DEVICE_MANF_CARD() macro
pcmcia_device structure 2nd
pcmcia_driver structure 2nd
pcmcia_get_first_tuple() function
pcmcia_get_tuple_data() function
pcmcia_parse_tuple() function
pcmcia_register_driver() function 2nd
pcmcia_request_irq() function
pcmcia_unregister_driver() function
pcmciautils package
pcspkr_event() function
pda_mtd_probe() function
pdflush kernel thread
Pentium TSC (Time Stamp Counter)
percent sign (%)
performance
  network throughput
    driver performance
    overview
    protocol performance
  performance governor
Peripheral Component Interconnect [See PCI (Peripheral Component Interconnect).]
peripherals
  choosing
  peripheral controllers
permanent virtual circuits (PVCs)
personal area network (PAN)
personal identification numbers (PINs)

PHY (physical layer) transceivers
PIBS bootloader
Pico-IrDA
PINs (personal identification numbers)
PIO (programmed I/O)
pipes 2nd
placement plots
platform_add_devices() function 2nd
platform_device_register() function
platform_device_register_simple() function 2nd 3rd
platform_device_unregister() function 2nd
platform_device register() function
platform_device structure 2nd
platform_driver_register() function 2nd
platform_driver_unregister() function
platform_driver structure 2nd
platform drivers
Plug-and-Play (PnP)
PMAC (Power Management and Audio Component)
PnP (Plug-and-Play)
PoE (Power over Ethernet)
point-of-sale (POS)
Point-to-Point Protocol (PPP) 2nd
pointers
poll() method 2nd 3rd
poll_table structure 2nd
poll_wait() function 2nd
polling in char drivers
populating URBs
port_data_in() function
port_data_out() function
portability of code
port char device
porting kernels
ports
  kgdb ports
  parallel port communication
  parallel port LED board
    controlling with sysfs
    led.c driver
  serial ports
  USB_UART ports
POS (point-of-sale)
post-handlers (kprobes)
power management
Power Management and Audio Component (PMAC)
Power over Ethernet (PoE)
PowerPC bootloaders
powersave governor
ppdev driver 2nd
PPP (Point-to-Point Protocol) 2nd
pppd daemon
pre-handlers (kprobes)
preempt_disable() function
preempt_enable() function
preemption counters
preprocessed source code, generating
printed circuit boards (PCBs)
printk() function 2nd 3rd
probe() function 2nd 3rd 4th
probes [See kprobes.]
probing
  EEPROM driver
  kprobes [See kprobes.]

network functions
telemetry card example
processes
contexts
init
kernel processes [See kernel threads.]
states
zombie processes
process filesystem [See procfs.]
processors, choosing
process scheduling (user mode drivers)
CFS (Completely Fair Scheduler)
O(1) scheduler
original scheduler
overview
procfs
documentation
reading with
example
large procfs reads
seq files
profiling
Bluetooth
gprof
OProfile
cache misses, counting
opcontrol
opreport
overview
programmed I/O (PIO)
protected mode 2nd
PS/2 mouse
ps command
pseudo char drivers
pseudo terminals (PTYs)
psmouse_protocol structure 2nd
psmouse structure
PTR_ERR() function
ptrace utility
pty.c driver
PTYs (pseudo terminals)
public domain software
pulse code modulation (PCM)
pulse-width modulator (PWM) units
PVCs (permanent virtual circuits)
PWM (pulse-width modulator) units

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

QoS (quality of service)
Qtronix infrared keyboard driver
quality of service (QoS)
Quarter VGA (QVGA)
queries, CFI-compliant flash
queues
    active queues
    expired queues
    overview
    run queues
    work queues 2nd 3rd
QVGA (Quarter VGA)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

race conditions
radio
    amateur radio
    RF (Radio Frequency) chips
    RFCOMM (Radio Frequency Communication)
    RFID (Radio Frequency Identification) transmitters
RAID (redundant array of inexpensive disks)
raise_softirq() function 2nd
random char device
random number generator
RapidIO
    Fibre Channel
    iSCSI (Internet SCSI)
RAS (reliability, availability, serviceability)
rc.sysinit file
RCU (Read-Copy Update)
RDMA (Remote DMA)
rdtsc() function
read() method
READ_CAPACITY command
Read-Copy Update (RCU)
read_lock() function
read_lock_irqrestore() function 2nd
read_lock_irqsave() function 2nd
read_seqbegin() function
read_seqlock() function
read_seqretry() function
read_sequnlock() function
read_unlock() function
reader-writer locks
reading data
    CMOS driver
    with procfs
        example
        large procfs reads
        seq files
readme_proc() function
    arguments
    example
    large procfs reads
    large proc reads
    seq files
read paths
readv() function
real mode 2nd
real time (-rt) patch 2nd
Real Time Clock (RTC) 2nd
Real Time Transport Protocol (RTP)
receive_buf() function
receive path (NICs)
receptacles (USB)
RedBoot 2nd
redundant array of inexpensive disks (RAID)
reference designators
register_blkdev() function 2nd

register_chrdev() function
register_die_notifier() function
register_inetaddr_notifier() function
register_jprobes() function
register_kretprobes() function
register_netdev() function 2nd
register_netdevice_notifier() function
registered protocol families
registering
    jprobe handlers
    kprobe handlers
    map drivers
    modem functions
    network functions
    PCMCIA client drivers
    platform drivers
    return probe handlers
    UART drivers
    user mode helpers
register layout
    audio hardware
    char drivers
    myblkdev storage controller
    USB_UART
release() method 2nd
release_firmware() function
release_region() function 2nd
reliability, availability, serviceability (RAS)
Remote DMA (RDMA)
remove() function
remove_wait_queue() function 2nd
reporting (ECC) [See ECC (error correcting code) reporting.]
request() method
request_firmware() function 2nd
request_irq() function 2nd 3rd 4th
request_mem_region() function 2nd
request_queue structure 2nd
request_region() function 2nd 3rd
requests, interrupt [See IRQs (interrupt requests).]
request structure 2nd
Request To Send (RTS)
response times (user mode drivers)
resume() function
return probes (kretprobes)
RF (Radio Frequency) chips
RFCOMM (Radio Frequency Communication) 2nd
RFID (Radio Frequency Identification) transmitters
rjcomm.ko
rmb() function
rmmod command
roller_analyze() function
roller_capture() function
roller_interrupt() function
roller mouse device example
roller_mouse_init() function
roller wheel device example
    edge sensitivity
    free_irq() function
    overview
    request_irq() function
    roller interrupt handler
    softirqs
    tasklets
    wave forms generated by

rootfs
  compact middleware
  NFS-mounted root
  obtaining
  overview
root hubs
Rose
rq_for_each_bio() function 2nd
RS-485
rs_open() function
–rt (real time) patch 2nd
RTC (Real Time Clock) 2nd 3rd 4th
rtc.c driver
rtc_class_ops structure 2nd
rtc_device_register() function 2nd
rtc_device_unregister() function 2nd
rtc_interrupt() function
RTP (Real Time Transport Protocol)
RTS (Request To Send)
run_umode_handler() function
runltp script
running state (threads)
run queues
rwlock_t structure

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

SAMPLING_RATE_REGISTER
SANs (storage area networks)
SAP (SIM Access Profile)
SAS (Serial Attached SCSI)
SATA (Serial ATA)
SBEs (single-bit errors)
SBP2 (Serial Bus Protocol 2)
scatter-gather
Scatterlist structure
sched_getparam() function
sched_param structure
sched_setscheduler() function 2nd
schedule() function
schedule_timeout() function 2nd 3rd
schedulers, I/O
scheduling processes [See process scheduling.]
SCIs (system control interrupts)
SCLK (Serial CLocK) 2nd
SCO (Synchronous Connection Oriented)
sco.ko
screen blanking
scripts
  build scripts
  runltp
  scripts directory
  sensors-detect
SCSI (Small Computer System Interface) 2nd
scsi_add_host() function
SCSI Generic (sg)
SD (Secure Digital) cards
SD/MMC
SDA (Serial Data)
SDP (Service Discovery Protocol)
SECTOR_COUNT_REGISTER
SECTOR_NUMBER_REGISTER
sectors 2nd
Secure Digital (SD) cards
security directory
SEEK_CUR command
SEEK_END command
SEEK_SET command
seek operation (CMOS driver)
seek times
select() method
Self-Monitoring, Analysis, and Reporting Technology (SMART)
semaphore structure 2nd
sensing data availability (char drivers)
  fasync() function
  overview
  select()/poll() mechanism
sensors-detect script
seq files
  advantages
  documentation
  large procfs reads

NVRAM drivers, updating
overview
seqlocks
sequence locks
serial_cs Card Services driver
serial8250_register_port() function
Serial ATA (SATA)
Serial Attached SCSI (SAS)
Serial Bus Protocol 2 (SBP2)
Serial CLocK (SCLK) 2nd
serial communication
Serial Data (SDA)
serial drivers
cell phone device example
claiming/freeing memory
CPLD (Complex Programmable Logic Device)
overview
platform drivers
SoC (System-on-Chip)
USB_UART driver
USB_UART ports
USB_UART register layout
data structures
layered architecture
line disciplines (touch controller device example)
changing
compiling
connection diagram
flushing data
I/O Control
open/close operations
opening
overview
read paths
unregistering
write paths
overview
sources
summary of kernel programming interfaces
TTY drivers
UART drivers
registering
uart_driver structure
uart_ops structure
uart_port structure
Serial Line Internet Protocol (SLIP)
serial PCMCIA
Serial Peripheral Interface (SPI) 2nd
serial ports
serio
serio_register_port() function
serport
Service Discovery Protocol (SDP)
service set identifiers (SSIDs)
Session Initiation Protocol (SIP)
set_bit() function
set_capacity() function 2nd
set_current_state() function 2nd
set_termios() function
set-top box (STB)
setitimer() function
sg (SCSI Generic)
SG_IO command
sg_io_hdr_t structure 2nd

sg3_utils package
short delays
showkey utility
SIG (Bluetooth Special Interest Group)
sigaction() function
signal_pending() function 2nd
SIGs (Special Interest Groups)
silk screens
SIM Access Profile (SAP)
simple_map_init() function
simple_map_write() function
simulating mouse movements
single-bit errors (SBEs)
single_open() function
SIP (Session Initiation Protocol)
sk_buff structure 2nd 3rd
skb_clone() function 2nd
skb_put() function 2nd
skb_release_data() function
skb_reserve() function 2nd
skbuff_clone() function
slave addresses
slaves
SLIP (Serial Line Internet Protocol)
SLOF bootloader
Small Computer System Interface (SCSI) 2nd
SMART (Self-Monitoring, Analysis, and Reporting Technology)
SMBus [See also I²C.]
  data access functions
  definition
  overview
SMIs (system management interrupts)
SMP (Symmetric Multi Processing) 2nd
snd_ac97_codec module
snd_card_free() function 2nd
snd_card_new() function 2nd
snd_card_proc_new() function 2nd
snd_card_register() function 2nd
snd_card structure
snd_ctl_add() function 2nd
snd_ctl_elem_id_set_interface() function
snd_ctl_elem_id_set_numid() function
snd_ctl_elem_info structure
snd_ctl_elem_write() function
snd_ctl_new1() function 2nd
snd_ctl_open() function
snd_device_new() function
snd_intel8x0 driver
snd_kcontrol_new structure
snd_kcontrol structure
snd_pcm_hardware structure
snd_pcm_lib_malloc_pages() function 2nd
snd_pcm_lib_preallocate_pages_for_all() function 2nd
snd_pcm_new() function 2nd
snd_pcm_ops structure 2nd
snd_pcm_runtime structure
snd_pcm_set_ops() function 2nd
snd_pcm_substream structure
snd_pcm structure
SoC (System-on-Chip)
sockets
  buffers
  Netlink sockets
  UNIX-domain sockets

softdogs
softirqs
  compared to tasklets
  definition
  ksoftirqd/0 kernel thread
softlockup_tick() function
soft lockups
software RAID
sound [See audio.]
sources
  audio drivers
  block drivers
  char drivers
  input drivers
  Inter-Integrated Circuit Protocol
  kdump
  kernels
  kexec
  kprobes
  MTD 2nd
  NICs (network interface cards)
  PCI
  PCMCIA
  serial drivers
  source tree layout
  USB (universal serial bus)
  user mode drivers
source tree layout
  directories
  navigating
South Bridge system
spaces (ACPI)
spare area (NAND chip drivers)
Special Interest Groups (SIGs)
speeds (USB)
SPI (Serial Peripheral Interface) 2nd
spi_asaync() function
spi_async() function 2nd
spi_butterfly driver
spi_device structure 2nd
spi_driver structure
spi_message_add_tail() functions
spi_message_init() functions
spi_message structure
spi_register_driver() function 2nd
spi_sync() function 2nd
spi_transfer structure
spi_unregister_driver() function
spin_lock() function 2nd 3rd
spin_lock_bh() function
spin_lock_init() function
spin_lock_irqsave() function
spin_unlock() function 2nd
spin_unlock_bh() function
spin_unlock_irqsave() function
spinlock_t structure
spinlocks
SSID (service set identifier)
ssize_t aio_read() function
ssize_t aio_write() function
start() function
start_kernel() function 2nd
start_tx() function
states of kernel threads

STATUS_REGISTER 2nd
STB (set-top box)
stop() function
stopped state (threads)
storage area networks (SANs)
storage controller [See myblkdev storage controller.]
storage_probe() function
storage technologies
    ATAPI (ATA Packet Interface)
    IDE (Integrated Drive Electronics)
    libATA
    MMC (MultiMediaCard)
    PCMCIA/CF
    RAID (redundant array of inexpensive disks)
    SATA (Serial ATA)
    SCSI (Small Computer System Interface)
    SD (Secure Digital) cards
    summary of
strace utility
streaming DMA access methods
struct e820map
structures [See *specific structures*.]
submit_work() function
submitting
    URBs for data transfer
    work to be executed later
subversion
Super I/O chips
Super Video Graphics Array (SVGA)
suspend() function
SVCs (switched virtual circuits)
SVGA (Super Video Graphics Array)
SVGAlib
swap space
switched virtual circuits (SVCs)
Symmetric Multi Processing (SMP) 2nd
synaptics_init() function
synaptics_process_byte() functions
synchronization
    completion functions
    kthread helpers
    SCO (Synchronous Connection Oriented)
    synchronous DMA
    synchronous interrupts
/sys/devices/system/edac/ directory
sysdiag utility
sysfs 2nd
sysfs_create_dir() function
sysfs_create_file() function
sysfs_create_group() function
sysfs_remove_group() function
sysfsutils package
SYSLINUX
syslog() function
System-on-Chip (SoC)
system control interrupts (SCIs)
System Management Bus [See SMBus.]
system management interrupts (SMIs)
system memory map
    copying
    obtaining
SystemTap

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

tables, nand_flash_ids[]
tail field (sk_buff structure)
TASK_INTERRUPTIBLE state
TASK_RUNNING state
TASK_STOPPED state
TASK_TRACED state
TASK_UNINTERRUPTIBLE state
tasklet_disable() function 2nd
tasklet_disable_nosync() function 2nd
tasklet_enable() function 2nd
tasklet_init() function 2nd
tasklet_schedule() function 2nd
tasklet_struct structure
tasklets
tele_device_t structure
tele_disconnect() function
tele_open() function
tele_probe() function
tele_read() function
tele_write() function
tele_write_callback() function
telemetry card example
    data transfer
    driver initialization
    pci_device_id structure
    probing and disconnecting
    register access
    register space
templates, libusb programming template
test_and_set_bit() function
test_bit() function
testing
    LTP (Linux Test Project)
    test equipment
    test infrastructure
TFT (Thin Film Transistor)
TFTP embedded devices
Thin Film Transistor (TFT)
threads [See kernel threads.]
throughput
    driver performance
    overview
    protocol performance
Thttpd
time() function
time_after() function
time_after_eq() function
time_before() function
time_before_eq() function
timer_func() functions
timer_list structure
timer_pending() function 2nd
timers
    HZ
    jiffies

long delays
overview
RTC (Real Time Clock)
short delays
TSC (Time Stamp Counter)
watchdog timer
Time Stamp Counter (TSC) 2nd
timeval structure
TinyTP (Tiny Transport Protocol)
TinyX
tool chains
Torvalds, Linus
touch controller
compiling
connection diagram
flushing data
I/O Control
open/close operations
opening
read paths
write paths
touchpads
touch screens
trace daemon
traced state (threads)
tracereader
tracevisualizer
tracing
LTT (Linux Trace Toolkit)
components
events
LTTng
LTTV (Linux Trace Toolkit Viewer)
trace dumps
overview
trackpoints
transactions (I^2C)
transceivers (USB)
transfer [See data transfer.]
Transistor-Transistor Logic (TTL)
transmit paths (NICs)
trojan_function() function
TROUBLED_DS environmental variable
TSC (Time Stamp Counter) 2nd
tsdev driver
TTL (Transistor-Transistor Logic)
tty.c driver
tty_buffer structure 2nd
tty_bufhead structure 2nd
tty_driver structure 2nd
TTY drivers
tty_flip_buffer_push() function 2nd
tty_flip_buffer structure
tty_insert_flip_char() function 2nd 3rd
tty_ldisc structure 2nd
tty_open() function
tty_register_device() function
tty_register_driver() function 2nd
tty_register_ldisc() function
tty_struct structure 2nd
tty_unregister_driver() function
tty_unregister_ldisc() function
TUN/TAP device driver
TUN network driver

tuple_t structure 2nd

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

U-Boot
uart_add_one_port() function 2nd 3rd
uart_driver structure 2nd
UART (Universal Asynchronous Receiver Transmitter) drivers 2nd
  cell phone device example
    claiming/freeing memory
    CPLD (Complex Programmable Logic Device)
    overview
    platform drivers
    SoC (System-on-Chip)
    USB_UART driver
    USB_UART ports
    USB_UART register layout
  registering
  RS-485
  uart_driver structure
  uart_ops structure
  uart_port structure
uart_ops structure 2nd
uart_port structure 2nd
uart_register_driver() function 2nd 3rd
uart_unregister_driver() function
UCEs (uncorrectable errors)
uClibc
uClinux
UDB
  class drivers
  debugging
udelay() function 2nd
udev
  on embedded devices
  PCMCIA
udevmonitor
udevsend
UHCI (Universal Host Controller Interface)
UIO (Userspace IO)
uIP
UML (User Mode Linux)
uncorrectable errors (UCEs)
uninterruptible state (threads)
Universal Asynchronous Receiver Transmitter [See UART (Universal Asynchronous Receiver Transmitter) drivers.]
Universal Host Controller Interface (UHCI)
universal serial bus [See USB (universal serial bus).]
UNIX-domain sockets
unlikely() function 2nd
unregister_blkdev() function
unregister_chrdev_region() function
unregister_netdev() function
unregister_netdevice_notifier() function
up() function
up_read() function
up_write() function
updating
  BIOS

NVRAM drivers
urandom char device
URBs (USB Request Blocks)
urb structure 2nd
USB (universal serial bus)
addressing
Bluetooth 2nd
bus speeds
class drivers
HIDs (human interface devices)
mass storage
overview
USB-Serial
compared to I²C and PCI
data structures
descriptors
pipes
tables of
URBs (USB Request Blocks)
usb_device structure
embedded drivers
on embedded systems
endpoints
enumeration
gadget drivers
host controllers
illustration of Linux-USB subsystem
kernel programming interfaces, table of
Linux-USB subsystem architecture
mice
OTG controllers
overview
receptacles
sources
telemetry card example
data transfer
driver initialization
pci_device_id structure
probing and disconnecting
register access
register space
transceivers
transfer types
URBs (USB Request Blocks)
usbfs virtual filesystem
USB Gadget project
USB-Serial
usb-serial.c driver
usb_[control|interrupt|bulk]_msg() function
usb_[rcv|snd][ctrl|int|bulk|isoc]pipe() function 2nd
usb_alloc_urb() function 2nd
usb_buffer_alloc() function
usb_buffer_free() function
usb_bulk_msg() function
usb_bus structure
usb_close() function
usb_config_descriptor structure 2nd
usb_control_msg() function 2nd 3rd
usb_ctrlrequest structure
usb_deregister() function
usb_deregister_dev() function
usb_dev_handle structure
USB_DEVICE() macro
usb_device_descriptor structure 2nd

usb_device_id structure
usb_device structure 2nd 3rd
usb_driver structure
usb_endpoint_descriptor structure 2nd
usb_fill_bulk_urb() function 2nd
usb_fill_control_urb() function 2nd 3rd
usb_fill_int_urb() function 2nd
usb_find_buses() function
usb_find_devices() function
usb_find_interface() function
usb_free_urb() function 2nd
usb_gadget_driver structure 2nd
usb_gadget_register_driver() function 2nd
usb_get_intfdata() function 2nd
usb_init() function
usb_interface_descriptor structure 2nd
usb_open() function
usb_register() function 2nd
usb_register_dev() function
usb_serial_deregister() function
usb_serial_driver structure
usb_serial_register() function 2nd
usb_set_intfdata() function 2nd
usb_submit_urb() function 2nd
usb_tele_init() function
USB_UART
USB_UART driver
    code listing
    register layout
USB_UART ports
usb_uart_probe() function
usb_uart_rxint() function
usb_uart_start_tx() function
usb_unlink_urb() function 2nd
usbfs virtual filesystem 2nd
USB Gadget project
usbhid driver
usbhid USB client driver
USB keyboards
usbmon command
USB Request Blocks (URBs)
usbserial drivers
user mode drivers
    data structures
    I/O regions
        accessing
        dumping bytes from
    memory regions, accessing
    parallel port LED boards, controlling
    process scheduling
        CFS (Completely Fair Scheduler)
        O(1) scheduler
        original scheduler
        overview
    response times
    sg (SCSI Generic)
    sources
    UIO (Userspace IO)
    usbfs virtual filesystem
    user mode I^2C
    user space library functions
    when to use
user mode helpers
User Mode Linux (UML)

User Modules
block device emulation
char device emulation
definition
JFFS (Journaling Flash File System)
MTD-utils
overview
YAFFS (Yet Another Flash File System)
user space drivers [See user mode drivers.]
userspace governor
Userspace IO (UIO)
user space library functions
usr directory
UU_READ_DATA_REGISTER
UU_STATUS_REGISTER
UU_WRITE_DATA_REGISTER

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

V2IP (Video-and-Voice over IP)
variables
    loops_per_jiffy 2nd 3rd
    xtime
VCI (virtual circuit identifier)
verify checksum command (ioctl)
version control
Very high speed integrated circuit Hardware Description Language (VHDL)
vesafb (video frame buffer driver)
VFS (Virtual File System) 2nd
vfs_readdir() function
VGA (Video Graphics Array)
VHDL (Very high speed integrated circuit Hardware Description Language
video
    cabling standards
    controllers
    embedded drivers
    VGA (Video Graphics Array)
    video frame buffer driver [See vesafb (video frame buffer driver).]
Video-and-Voice over IP (V2IP)
video1394 driver
virtual addresses
virtual circuit identifier (VCI)
Virtual File System (VFS) 2nd
virtual mouse device example
    gpm (general-purpose mouse)
    simulating mouse movements
    vms.c input driver
Virtual Network Computing (VNC)
virtual path identifier (VPI)
virtual terminals (VTs)
Vital Product Data (VPD)
vmalloc() function 2nd
vmlinux kernel image
vms.c application
vms_init() function
VNC (Virtual Network Computing)
VoIP (Voice over Internet Protocol)
VOLUME_REGISTER
VPD (Vital Product Data)
VPI (virtual path identifier)
vt.c driver
VTs (virtual terminals)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

w1 bus
w1_family_ops structure
w1_family structure
wait_event_timeout() function 2nd
wait_for_completion() function 2nd
wait_queue_t structure
wait queues [See queues.]
wake_up_interruptible() function 2nd 3rd
wall time
watchdog timeout
watchdog timer
watchpoints
wd33c93_init() function
wear leveling
WiFi 2nd 3rd
WiMax
wireless
  trade-offs for
  WiFi 2nd 3rd
  Wireless Extensions
wmb() function 2nd
work, submitting to be executed later
work_struct structure 2nd
worker thread
workqueue_struct structure
work queues 2nd 3rd
write() method
write_lock() function
write_lock_irqrestore() function 2nd
write_lock_irqsave() function 2nd
write_seqlock() function
write_sequnlock() function
write_unlock() function
write_vms() function
write_wakeup() function
writev() function
writing
  CMOS driver
  input event drivers

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

x86 bootloaders
xf86SIGIO() function
Xf86WaitForInput() function
XGA (eXtended Graphics Array)
XIP (eXecute In Place)
xtime variable
X Windows

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

YAFFS (Yet Another Flash File System)

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

zero-page.txt file
zero char device
zero page
Zigbee
zombie processes
zombie state (threads)
ZONE_DMA
ZONE_HIGH
ZONE_NORMAL